MASTER EN SCIENCES MATHÉMATIQUES

Étude et implémentation du LISP

LE CHARLIER, Baudouin

Award date:
1973

Awarding institution:
Universite de Namur

LTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX - NAMUR

INSTITUT D'INFORMATIQUE

1972-1973

Etude et Implémentation du LISP

B. LECHARLIER

Mémoire préesenté pour l'obtention
du grade de Licencié et Maître en Informatique

JURY : J.-P. CARDINAEL

REMERCIEMENTS.

Nous tenons à remercier bien vivement Monsieur D. Ribbens, professeur d'informatique à l'université de Liège, pour l'aide précieuse qu'il nous a apportée lors de la conception et de la réalisation de l'interpréteur.

Toute notre gratitude va à Monsieur J.-P. Cardinael, directeur de ce mémoire, pour l'aide constante qu'il nous a apportée lors de la rédaction en discutant avec nous de la présentation théorique du langage.

TABLE DES MATIERES.

INTRODUCTION.

Dans ce travail, à côté de la description de l'implémentation d'un interpréteur LISP, nous avons tenté de faire une étude critique de ce langage en séparant, des notions sans originalité véritable, celles qui font du LISP un langage à part dans l'ensemble des langages de programmation évolués couramment utilisés.

La première partie étudie le LISP "pur" lequel contient toutes les caractéristiques vraiment intéressantes de ce langage, au point de vue théorique.

Le LISP est aussi un des seuls langages qui fasse un usage systématique d'un programme récupérateur pour la gestion de la mémoire. Pour cette raison, la quatrième partie, qui traite de ce problème, est une des plus importantes de ce travail.

La deuxième partie décrit les principales extensions apportées au LISP "pur" et donne la raison de leur introduction.

La troisième partie traite des programmes d'entrée/sortie et du superviseur. Elle contient surtout des détails d'implémentation.

Enfin, la cinquième partie contient deux exemples de traitements écrits en LISP et le résultat de leur exécution par notre interpréteur. Elle prouve le bon fonctionnement de celui-ci.

PREMIERE PARTIE : LE LISP "PUR".

Chapitre 1 : Les types de valeurs et les algorithmes primitifs.

§ 1. Types de valeurs en LISP .

Du point de vue de son champ d'application, le LISP est un langage de traitement de l'information non numérique, plus précisément, de l'information pouvant se traduire de façon plus ou moins commode par des listes. La dérivation formelle, la démonstration automatique de théorèmes sont des exemples de tels traitements.

Rappelons la définition d'une liste:
Une liste est une suite finie, peut-être vide, de symboles atomiques ou de listes, mise entre parenthèses.
Par symbole atomique, on entend en LISP: suite finie de lettres majuscules et de chiffres, commençant par une lettre.

Voici un exemple de liste:

((FØRTRAN,ALGØL,CØBØL)(PL1)). (1) (*)

Nous allons chercher à mettre en évidence un type de représentation des listes en mémoire d'ordinateur rendant les opérations de manipulation de listes les moins coûteuses possible. (Aux points de vue place mémoire et temps d'exécution.)

Un premier mode de représentation, évident, consisterait à ranger toute liste en mémoire sous forme de cases de mémoire contiguës, chaque case contenant la représentation interne du caractère correspondant de la liste. C'est la façon de représenter les chaînes de caractères dans des langages tels que PL/I. Dans cette hypothèse, la liste (1) occupe, en mémoire, au moins, 28 cases consécutives. Toutefois, cette manière de procéder ne serait guère féconde dans un langage de manipulation de listes, car elle rend très lour le traitement. Par exemple, considérons les deux listes:

((FØTRAN,ALGØL,CØBØL)(PL1)) et ((LISP,SLIP) (IPLV))

Si l'on veut créer la liste ayant pour éléments tous les éléments de ces deux listes, on devra, forcément, recopier ces éléments et réserver de la place mémoire pour y mettre la liste:

((FØRTRAN,ALGØL,CØBØL)(PL1)(LISP,SLIP)(IPLV))

(*) On convient de séparer les symboles atomiques, dans les listes, par des blancs ou des virgules. D'une façon générale, on admet qu'on ne modifie pas la nature d'une liste si on intercale entre ses éléments un nombre arbitraire de blancs et/ou de virgules. Ainsi, la liste (CAR,CDR) est équivalente à la liste (,,CAR,,CDR,).

La constitution de cette liste exige:

- réservation de place pratiquement égale à celle occupée par les listes initiales,

- recopiage quasi intégral des listes initiales dans la nouvelle zone.

Autrement dit, on gaspille de la place et du temps d'exécution. Nous allons, donc, chercher de meilleurs modes de représentation interne pour les listes; c'est cette démarche qui nous amènera à définir la S-expression, type de valeur le plus général en LISP comme un corollaire du choix de la "meilleure représentation interne" des listes.

Correspondance Arbre Liste.

On appelle arbre un ensemble A de n noeuds ($n \geq 1$) tel que:

- il existe un noeud privilégié appelé racine de l'arbre,

- l'ensemble des noeuds restants peut être partitionné en m ($m \geq 0$) sous-ensembles qui sont, eux-mêmes, des arbres: A_1, ... , A_m.

Les arbres A_1, ... , A_m sont appelés sous-arbres de A.

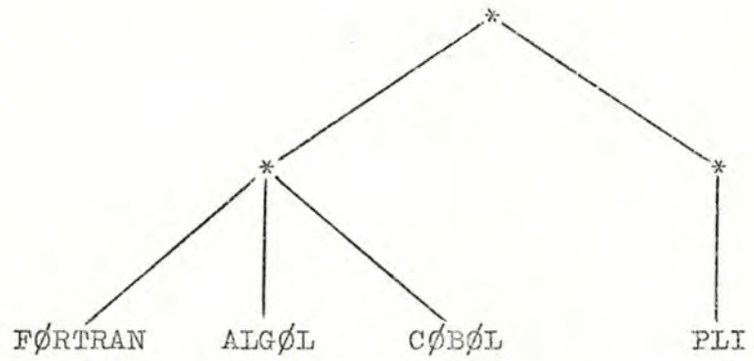
Les noeuds de A_1, ... , A_m sont appelés descendants de la racine de A, et les racines de A_1 ... , A_m sont appelées successeurs de la racine de A, enfin les noeuds qui n'ont pas de descendants sont appelés noeuds terminaux.

La représentation la plus habituelle d'un arbre est constituée par un ensemble de points, les noeuds, reliés entre eux par des traits symbolisant les liens entre les racines et leurs sous-arbres.
Ceci dit, considérons un type d'arbres particulier tel que leurs noeuds soient tous le symbole "*", sauf pour les noeuds terminaux qui peuvent être, soit des symboles atomiques, soit le symbole "*". Il est facile de se convaincre de ce que l'arbre de la figure 1 est équivalent à la liste (1) en tenant compte de la table de correspondance suivante:

ARBRE	LISTE
- noeud terminal:- atome	- atome.
- "*".	- liste vide.
- sous-arbre non réduit à un noeud terminal.	- sous-liste.

On pourrait se servir de cette équivalence pour imaginer un mode de représentation des listes en mémoire: il serait naturel, dans ce cas, que chaque noeud de l'arbre soit représenté par une cellule de mémoire. Les cellules représentant des "*" contiendraient une série de pointeurs vers leurs successeurs et les cellules représentant des symboles atomiques contiendraient les caractères de la représentation externe du symbole; toutefois, l'emploi de cellules de mémoire de longueur variable risque de poser des problèmes lors du traitement et nous allons voir une méthode permettant d'utiliser des cellules de longueur fixe.

Figure 1.



Arbre binaire.

Une structure, semblable à celle d'arbre, mais plus facile à implémenter est celle d'arbre binaire. On appelle arbre binaire,

soit l'ensemble vide,

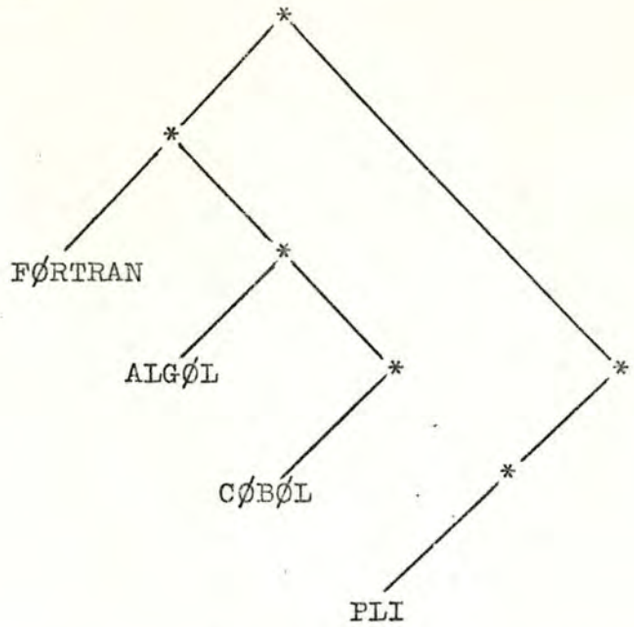
soit l'ensemble d'une racine et de deux arbres binaires disjoints appelés sous-arbre binaire gauche et sous arbre binaire droit.

En examinant conjointement les définitions d'arbre et d'arbre binaire, on se convaincra de ce que la structure d'arbre binaire n'est pas un cas particulier de la structure d'arbre, puisque, par exemple, l'ensemble vide est un arbre binaire, mais pas un arbre. Tout arbre a, en effet, au moins un noeud: sa racine.

Comme pour les arbres, nous ne considérerons que des arbres binaires dont les noeuds non terminaux sont des "*" et les noeuds terminaux sont soit des symboles atomiques, soit des "*".

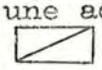
La figure 2 montre un exemple d'arbre binaire.

Figure 2.



L'implémentation de la structure d'arbre binaire est la suivante:

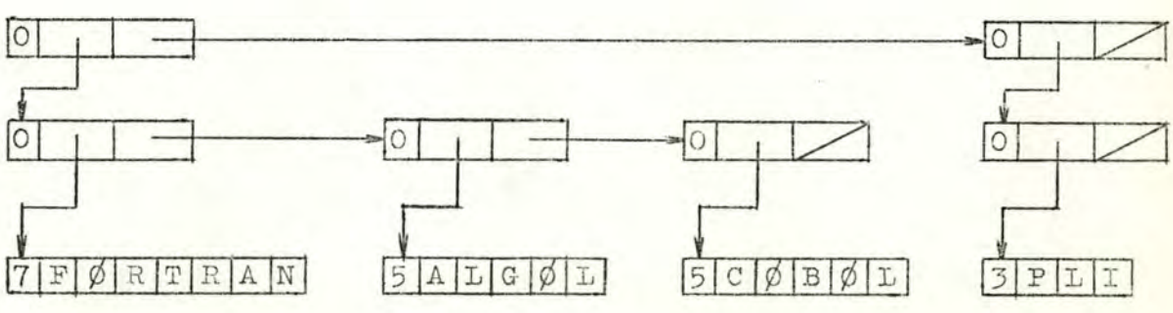
- un atome est représenté en mémoire par une adresse pointant vers une zone qui contient les caractères de sa représentation externe précédés d'un indicateur précisant que la zone qui suit contient des caractères et donnant la longueur de la zone.

- un arbre binaire vide est représenté par une adresse particulière que l'on désignera par le symbole  .

- un arbre binaire ayant pour racine une "*" est représenté par l'adresse d'une cellule mémoire contenant un indicateur et deux champs; l'indicateur précise que la zone qui suit contient deux adresses, les deux champs contiennent les représentations du sous-arbre binaire gauche et du sous arbre binaire droit.

En convenant de désigner l'adresse d'une zone de mémoire par une flèche pointant vers le début de cette zone, l'arbre binaire de la figure 2 donnera lieu en mémoire à l'assemblage de cellules de la figure 3.

Figure 3.



S-expression.

De même qu'on avait établi une équivalence entre la structure bidimensionnelle d'arbre et la structure linéaire de la liste, on peut associer, par introduction des symboles de ponctuation "(" , ")" , "." , une structure linéaire à la structure d'arbre binaire. Il s'agit de la structure de S-expression.

Une S-expression est
soit un symbole atomique,
soit une paire pointée, c'est à dire l'ensemble de deux S-expressions, séparées par un point et mises entre parenthèses.
L'équivalence est réalisée par la table de correspondance suivante:

Arbre binaire	S-expression
-arbre binaire vide.	-symbole atomique "NIL".
-arbre binaire atomique. (noeud terminal ≠ "*")	-symbole atomique.
-arbre binaire non atomique.	- paire pointée.
-sous-arbre binaire gauche.	- premier élément d'une paire pointée.
-sous-arbre binaire droit.	- second élément d'une paire pointée.

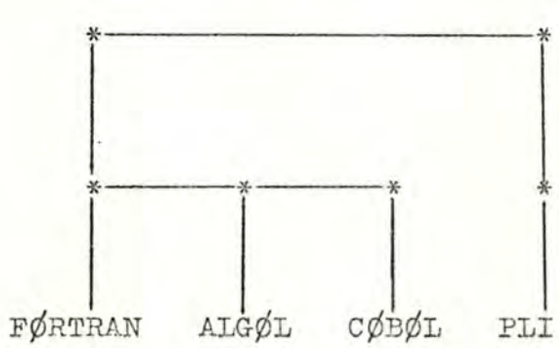
On peut remarquer que l'arbre binaire * correspond à la S-expression (NIL.NIL) .
D'après la table, la S-expression équivalente à l'arbre binaire de la figure 2 est :

((FØRTRAN.(ALGØL.(CØBØL.NIL))).((PLI.NIL).NIL)) (2) .

Représentation des arbres par des arbres binaires.

Faisons effectuer une rotation de 45 degrés à l'arbre binaire de la figure 2. Nous obtenons la figure suivante:

Figure 4.



Si nous fusionnons les "*" situées sur un même trait horizontal, nous retrouvons l'arbre de la figure 1; nous avons ainsi défini une correspondance entre les arbres et les arbres binaires.
Ceci nous permet, grâce aux équivalences établies plus haut de redéfinir une liste comme une S-expression d'un type particulier. Comme nous avons établi un mode de représentation interne pour les arbres binaires, c'est à dire pour les S-expressions, nous aurons, du même coup, un mode de représentation des listes.

Redéfinition des listes.

On appelle liste, une S-expression qui est
soit le symbole atomique "NIL",
soit une paire pointée de la forme

$$(\alpha_1.(\alpha_2.(\ \ldots \ (\alpha_n.\mathrm{NIL}) \ \ldots \ )))$$

où les α_i sont des S-expressions. De plus, on convient d'écrire un tel type de S-expression sous la forme

$$(\alpha_1,\alpha_2, \ \ldots \ ,\alpha_n) \ .$$

La S-expresion (2) n'est donc qu'une autre façon d'écrire la liste (1) . L'énoncé de toutes les équivalences vues plus haut le prouve. On peut aussi le voir directement en remarquant que :

(PLI.NIL) = (PLI)

(FØRTRAN.(ALGØL.(CØBØL.NIL))) = (FØRTRAN,ALGØL,CØBØL)

et donc,

(2) = ((FØRTRAN,ALGØL,CØBØL).((PLI).NIL))

= ((FØRTRAN,ALGØL,CØBØL)(PLI)) = (1) .

Le développement qui précède, montre que la notion du S-expression a été introduite, principalement, en vue de permettre une implémentation efficiente des structures de liste. Reste à montrer que cette implémentation est effectivement efficiente. Nous nous contenterons de la comparer à la méthode élémentaire décrite à la page I.1 .

Prenons l'exemple des deux listes,

((FØRTRAN,ALGØL,CØBØL)(PLI)) et ((LISP,SLIP)(IPLV)) .

Faisons l'hypothèse que la place mémoire occupée par une adresse est le double de celle occupée par un caractère. Dans ces conditions, cherchons a construire la liste

(FØRTRAN,ALGØL,CØBØL,PLI,LISP,SLIP,IPLV)

formée des symboles atomiques des deux listes.
Par la méthode de la page I.1, il faut réserver un nombre de cellules contiguës suffisant pour y loger la nouvelle liste. Cela exige:

- un examen des deux listes initiales pour calculer le nombre de caractères de la nouvelle liste (40) .

- le choix d'une zone de mémoire de longueur égale à 40 caractères,

- le réexamen des deux listes pour recopier les symboles atomiques dans la nouvelle zone.

Par la méthode que nous avons choisie, il suffit d'explorer tous les noeuds des arbres binaires représentant les deux listes initiales. Chaque fois qu'on arrive à un noeud qui est un symbole atomique, on choisit une cellule de mémoire constituée de deux champs d'une adresse (cellule équivalente à 4 caractères). Dans le premier champ, on met l'adresse du noeud atomique, dans le second champ, on met l'adresse ⊠ si c'est la première fois qu'on choisit une cellule, et sinon, on y met l'adresse de la dernière cellule choisie.

Outre le fait qu'on utilise moins de place mémoire avec la seconde méthode, cette place est répartie en 7 cellules de dimensions fixes, qui une fois devenues inutiles, pourront être réutilisées directement, alors que l'utilisation de zones de longueur variable, par la première méthode pose des problèmes lorsqu'on veut réutiliser des zones périmées pour y mettre d'autres listes de longueurs différentes (Compactage) .
De plus, le traitement à effectuer avec la seconde méthode est plus simple: il s'agit seulement de parcourir un arbre binaire, en suivant des pointeurs, tester l'atomicité d'une zone, recopier des adresses.
Dans le traitement par la première méthode, il faut examiner les listes caractère par caractère, compter les caractères, examiner chaque caractère pour voir si c'est "(" ou ")" ou "," ou " " ou une lettre.
La copie d'un atome, par la première méthode, implique le transfert d'une zone de longueur variable; par la seconde méthode, il suffit de copier une adresse.

Définition des types de valeurs en LISP.

Nous pouvons, maintenant, donner une définition des types de valeurs traitées par le LISP, sans qu'il soit besoin d'expliquer davantage le pourquoi de cette définition.

- Toutes les valeurs traitées par le LISP sont des S-expressions.

- Une S-expression est
  soit un atome, c'est à dire une suite finie de lettres majuscules et de chiffres commençant par une lettre,
  soit une paire pointée, c'est à dire un être de la forme (<S-expression>.<S-expression>).

- Parmi les S-expressions, il en est de particulières, appelées listes. Une liste est
  soit le symbole atomique spécial "NIL" désigné aussi par "()" et appelé "liste vide",
  soit une paire pointée du type (<S-expression>.<liste>).

Il découle de cette définition que toute liste peut s'écrire sous la forme

$$(\alpha_1.(\alpha_2.(\ \ldots \ (\alpha_n.\mathrm{NIL}) \ \ldots \ )))$$

où les α_i sont des S-expressions; cependant, on convient de l'écrire plutôt

$$(\alpha_1, \ \ldots \ ,\alpha_n)$$

ou encore,

$$(\alpha_1 \ \ \ldots \ \alpha_n).$$

§ 2. Les algorithmes primitifs du LISP.

Dans tout langage de programmation, les notions de type de valeurs et d'algorithme primitif, ou primitive du langage sont fortement liées. En Algol 60 ,par exemple, on a les types entier et réel, et on a les opérateurs primitifs "x" , "+" , "-" , "·/·" , "/" , "↑" qui, combinés, permettent d'effectuer un traitement quelconque sur des valeurs de type entier ou réel.

En LISP "pur", on a 5 algorithmes primitifs qui constituent la base du traitement des S-expressions. Ces algorithmes sont définis comme des fonctions admettant un ou deux arguments qui sont, évidemment, des S-expressions. Voici la définition de ces fonctions.

Les fonctions "car" et "cdr" : sélection des éléments d'une liste.

Convenons de désigner une S-expression par une lettre minuscule; soit alors x , une S-expression quelconque. Si x est une paire pointée $(\alpha.\beta)$,

$$car[x] \quad \text{est} \quad \alpha \ ,$$

$$cdr[x] \quad \text{est} \quad \beta$$

et, si x est un symbole atomique, la valeur de l'application d'une de ces fonctions à x est indéfinie.

Dans le cas d'une S-expression écrite sous forme de paire pointée, on voit bien que "car" et "cdr" sélectionnent, respectivement, la première et la seconde S-expression de la paire. Examinons leur action dans le cas d'une liste.

Soit $x = (\alpha_1, \ \ldots \ , \alpha_n)$,une liste. On a :

$$\begin{aligned} car[x] &= car[(\alpha_1, \ \ldots \ , \alpha_n)] \\ &= car[\underbrace{(\alpha_1}_{\alpha} . \underbrace{(\alpha_2 . (\ \ldots \ (\alpha_n . NIL) \ \ldots))}_{\beta})] \\ &= \alpha_1 \quad , \\ cdr[x] &= cdr[(\alpha_1 . (\alpha_2 . (\ \ldots \ (\alpha_n . NIL) \ \ldots)))] \\ &= (\alpha_2 . (\ \ldots \ (\alpha_n . NIL) \ \ldots \ )) \\ &= (\alpha_2, \ \ldots \ , \alpha_n) \ . \end{aligned}$$

On voit que "car" sélectionne le premier élément d'une liste et "cdr" la liste des éléments restants. On en déduit, immédiatement, que, pour sélectionner le k-ième élément d'une liste

$$(\alpha_1, \ldots, \alpha_n) \quad ,$$

il faut d'abord lui appliquer k-1 fois "cdr" pour obtenir la liste

$$(\alpha_k, \ldots, \alpha_n) \quad ,$$

puis une fois "car" pour obtenir α_k.

Exemples.

- car[NIL]
- cdr[()]

} sont indéfinis.

- car[(ALGØL.CØBØL)] = ALGØL
- cdr[(ALGØL.CØBØL)] = CØBØL
- cdr[(ALGØL,CØBØL)] = (CØBØL)
- cdr[(ALGØL)] = () = NIL .

La fonction "cons" : construction d'une paire pointée.

Soient x et y ,deux S-expressions quelconques; alors,

cons[x;y] est la paire pointée (x.y) .

Cette fonction réalise, en quelque sorte, l'opération inverse des deux fonctions précédentes. Plus précisément, si x est une paire pointée, on peut calculer :

t = car[x] ,

u = cdr[x]

et on a, alors :

cons[t;u] = x .

Si l'on veut construire une liste dont les éléments sont les n S-expressions $\alpha_1, \ldots, \alpha_n$, on peut procéder de la façon suivante:

$$\beta_1 = \text{cons}[\alpha_n;\text{NIL}] = (\alpha_n.\text{NIL}) = (\alpha_n) \quad ,$$

$$\beta_2 = \text{cons}[\alpha_{n-1};\beta_1] = (\alpha_{n-1}.(\alpha_n)) = (\alpha_{n-1},\alpha_n) \quad ,$$

... ...

$$\beta_n = \text{cons}[\alpha_1;\beta_{n-1}] = (\alpha_1.(\alpha_2, \ldots ,\alpha_n)) = (\alpha_1, \ldots ,\alpha_n) \quad .$$

Les prédicats "atom" et "eq" : valeurs logiques en LISP.

Lorsqu'on effectue un traitement sur des valeurs, il est toujours nécessaire de pouvoir tester la nature de certains résultats intermédiaires pour décider de la suite du traitement. Nous indiquerons, dans la suite, les règles utilisées en LISP pour programmer un traitement complet, mais nous définissons, ici, les deux prédicats "atom" et "eq" permettant d'effectuer des "tests élémentaires" sur des S-expressions.

Un prédicat est une fonction "logique" , elle ne pèut prendre que deux valeurs : "vrai" ou "faux". En LISP, on décide que

le symbole atomique "F" représente la valeur logique "faux" et que

le symbole atomique "T" représente la valeur logique "vrai".

Ceci dit, on comprendra, sans peine, la signification des définitions suivantes.

Si x et y sont des S-expressions, alors,

atom[x] est T si x est un symbole atomique,

atom[x] est F si x est une paire pointée;

si x et y sont des symboles atomiques,

eq[x;y] est T si $x = y$,

eq[x;y] est F si $x \neq y$,

sinon, la signification de eq[x;y] est indéfinie.

Exemples.

- atom[NIL] = T
- atom[( )] = T
- atom[F] = T
- atom[((FØRTRAN,ALGØL,CØBØL)(PLI))] = F
- eq[T;F] = F
- eq[NIL;()] = T
- eq[(A.(B.C));(A.(B.C))] est indéfini.

Chapitre 2 : Le "style" de programmation en LISP.

§ 1. Introduction.

En plus des algorithmes primitifs qui définissent les traitements élémentaires des valeurs, n'importe quel langage de programmation possède un certain nombre de mécanismes permettant de les combiner entre eux de façon à constituer des algorithmes d'une complexité, en principe, illimitée. Alors que les types de valeurs et leurs primitives associées définissent le "champ d'application" du langage, on peut dire que l'ensemble de ces mécanismes définit un "style" de programmation.

A ce point de vue, le LISP se distingue fondamentalement de la plupart des langages habituels tels que FØRTRAN, Algol, PL/I. Ceux-ci peuvent être qualifiés de "séquentiels" parce que un programme écrit dans ces langages peut être décomposé en une suite d'actions qui seront exécutées l'une après l'autre au cours du temps. Le LISP, par contre, est qualifié de "fonctionnel" parce que l'ensemble du programme et de ses données y revêt l'apparence de l'application d'une fonction (le programme) à des arguments constants (les données). Cela veut dire qu'on définit globalement le traitement à effectuer sans le décomposer en actions dont on fixe l'ordre d'exécution.

Un programme, écrit en LISP, est donc une construction du type

$$f[\alpha_1; \ldots ;\alpha_n] \quad ,$$

où les α_i sont des S-expressions. Une telle construction est appelée "forme". Les notions de forme et de fonction jouent un rôle très important en LISP. C'est pourquoi nous discuterons dans le § 2 des relations qui les lient, indépendamment de tout langage de programmation, de façon à éviter toute confusion à leur sujet.

§ 2. Les formes et les fonctions.

Les notions de forme et de fonction sont intimement liées. On peut dire que:

- une forme est l'application d'une fonction (dont on connaît la définition) à des arguments déterminés, tandis que

- une fonction peut être définie par l'association d'une liste de variables et d'une forme dans laquelle ces variables peuvent figurer.

Les variables de la liste sont dites "liées" ou encore "muettes", tandis que les variables apparaissant dans la forme mais pas dans la liste sont dites "libres". Il faut, pour que ces définitions ne constituent pas une pétition de principe, admettre qu'il existe des "formes primitives", ce sont les variables et les constantes, et des "fonctions primitives" appelées parfois "opérateurs": "x" , "+" , "-" , "/" , etc....

Donnons un exemple, utilisant les notations de l'algèbre moderne:

$$f: \mathbb{R} \longrightarrow \mathbb{R}$$

$$[x] \longmapsto 2x^2+3x+4$$

Au sens précisé plus haut, "f" est une fonction définie par association de la variable x et de la forme $2x^2+3x+4$. La signification de l'association $[x] \longmapsto 2x^2+3x+4$ est:

" Si on veut calculer la valeur de la fonction pour un nombre réel donné, il faut remplacer, dans la forme, toutes les occurences de x, par le nombre et évaluer le résultat". (Dans lequel, il n'y a plus que des constantes.) (règle 1)

La forme $2x^2+3x+4$ est définie, quant à elle, par combinaison des formes primitives "2" , "3" , "4" , "x" et des opérateurs primitifs "$*$" , "$\uparrow$" , "+" .

Remarquons, enfin, en examinant la règle 1 que le nom donné à une variable liée n'a pas d'importance, c'est à dire qu'on ne change pas la définition d'une fonction si on remplace le nom d'une variable liée par un autre symbole, sauf, toutefois, si ce symbole apparaît dans la forme initiale.
Ainsi les fonctions (1) et (2) sont équivalentes:

(1) $[x;y] \longmapsto ax^2 + bxy + cy^2$

(2) $[u;y] \longmapsto au^2 + buy + cy^2$ (on a remplacé x par u.)

(3) $[y;y] \longmapsto ay^2 + by^2 + cy^2 = (a+b+c)\, y^2$

(on a remplacé x par y "liée".)

(4) $[c;y] \longmapsto ac^2 + bcy + cy^2 = (ac + by + y^2)\, c$

(on a remplacé x par c "libre".)

Par contre, aucune des fonctions (3) ou (4) n'est équivalente à (1): la fonction (3) est une fonction à un seul argument, alors que (1) a deux arguments; la fonction (4) est une fonction du 3ème degré non homogène, alors que (1) est homogène du 2ème degré.

Nous allons, maintenant, étudier les mécanismes fournis par le LISP pour construire des formes et des fonctions. Comme un programme LISP est une forme particulière: $f[\alpha_1; \ldots ;\alpha_n]$ où les α_i sont des S-expressions "données", nous saurons, du même coup, écrire tous les programmes LISP possibles.

§3 . Construction des formes en LISP.

Le premier mécanisme du LISP pour la construction des formes est <u>la composition</u>:

Etant données, une forme du type $f[x_1; \ldots ;x_n]$ où les x_i sont des variables et n formes $\varepsilon_1, \ldots ,\varepsilon_n$, l'expression $f[\varepsilon_1; \ldots ;\varepsilon_n]$ est encore une forme.

<u>Exemple.</u>

- car[(A.B)], cdr[x] ,

  cdr[(A.B)], cons[x_1;x_2] sont des formes et, donc,

- cons[cdr[(A.B)]; car[(A.B)]] et

  cdr[cons[cdr[(A.B)];car[(A.B)]]] sont aussi des formes.

On peut par composition, obtenir des algorithmes assez compliqués, cependant, ces algorithmes seront toujours <u>statiques</u>, c'est à dire fixés une fois pour toutes, indépendamment des valeurs des arguments. En vue de pouvoir programmer des algorithmes qui tiennent compte de la nature des données à traiter, on introduit un second mécanisme:

<u>les formes conditionnelles</u>.

Etant données deux séries de formes $\varepsilon_1, \ldots ,\varepsilon_n$ et $\omega_1, \ldots ,\omega_n$, l'expression

$$[\varepsilon_1 \rightarrow \omega_1; \ldots ;\varepsilon_n \rightarrow \omega_n]$$

est aussi une forme, appelée forme conditionnelle et dont l'interprétation est la suivante: on cherche la première forme ε_i qui a la valeur "vrai", plus précisément dont la valeur n'est pas le symbole atomique "F", si une telle forme existe, la valeur de la forme conditionnelle est la valeur de ω_i, sinon sa valeur est indéfinie.

Deux remarques s'imposent:

- D'après la définition d'une forme conditionnelle, toute S-expression différente de "F" représente la valeur logique "vrai", dans la sémantique du LISP.

- Les formes ε_i ne peuvent prendre, en principe, que les valeurs "F" ou "T", c'est à dire que ce sont des formes prédicatives: elles renvoient des valeurs logiques.

Exemples de formes en LISP.

- $cdr[cons[cdr[(A.B)];car[(A.B)]]]$

$= cdr[cons[B;A]] = cdr[(B.A)] = A$.

- $[atom[x]\rightarrow[eq[x;NIL]\rightarrow NIL;T\rightarrow F];T\rightarrow cdr[x]]$

Cette forme est du type $[\varepsilon_1\rightarrow\omega_1;\varepsilon_2\rightarrow\omega_2]$

où $\varepsilon_1 = atom\ x$, $\varepsilon_2 = T$

$\omega_1 = [eq[x;NIL]\rightarrow NIL;T\rightarrow F]$, $\omega_2 = cdr[x]$.

Si nous remplaçons la variable x par la S-expression A, nous obtenons:

- $[atom[A]\rightarrow[eq[A;NIL]\rightarrow NIL;T\rightarrow F];T\rightarrow cdr[A]]$

$= [T\rightarrow[F\rightarrow NIL;T\rightarrow F];T\rightarrow ?]$

$= [F\rightarrow NIL;T\rightarrow F] = F$.

Nous verrons, dans la suite, un grand nombre d'exemples de formes conditionnelles.

§ 4. Définition des fonctions en LISP : les notations "λ" et "label"

En LISP "pur", les fonctions primitives sont, comme nous l'avons vu, au nombre de cinq: "car","cdr" , "cons", "atom" ,"eq". Toutes les autres fonctions doivent être définies d'une façon ou d'une autre par association d'une forme et d'une liste de variables.

Pour procéder à cette définition, on pourrait, dans un programme, écrire d'abord une expression du genre

$f: [x_1; \ldots ;x_n]\not\rightarrow \varepsilon$ où f est un identificateur

et ε la forme associée;

ensuite, si on veut calculer la valeur de la fonction pour des arguments $\alpha_1, \ldots ,\alpha_n$, on écrira $f[\alpha_1; \ldots ;\alpha_n]$. On a donc défini la fonction avant son emploi. En LISP, on décide de définir la fonction "f" au moment de son emploi et au lieu de procéder comme ci-dessus, on écrira directement:

$$\lambda[[x_1; \ldots ;x_n];\varepsilon][\alpha_1; \ldots ;\alpha_n].$$

L'expression $\lambda[[x_1; \ldots ;x_n];\varepsilon]$ définit la fonction et la désigne en même temps. (Elle joue le rôle de l'identificateur "f".) C'est la façon d'écrire, en LISP, "la fonction définie en associant la suite de variables $x_1, \ldots, x_n$ à la forme ε".

Voici, par exemple, la définition de la fonction qui sélectionne le troisième élément d'une liste:

$$\lambda[[x];car[cdr[cdr[x]]]] \quad .$$

Pour sélectionner le troisième élément de la liste (A,B,C) on peut donc écrire:

$$\lambda[[x];car[cdr[cdr[x]]]][(A,B,C)]$$

$$= car[cdr[cdr[(A,B,C)]]] \text{ (remplacement de x par sa valeur.)}$$

$$= car[cdr[(B,C)]]$$

$$= car[(C)]$$

$$= C$$

Ce mécanisme est, toutefois, insuffisant si l'on veut définir des fonctions de façon récursive. Considérons, par exemple, la fonction, nommée "fd" qui cherche le dernier élément d'une liste non vide. On peut écrire:

$$fd = \lambda[[x];[atom[cdr[x]]\rightarrow car[x];T\rightarrow fd[cdr[x]]]] \quad .$$

Cette expression signifie: "fd est la fonction d'un argument x,
qui regarde si x est une liste d'un seul élément,
(en effet, dans ce cas, $x = (\alpha_1) = (\alpha_1 .NIL)$ et
$atom[cdr[x]] = atom[NIL] = T$)
si oui, elle sélectionne son unique élément,
si non, on sélectionne la liste obtenue en supprimant le premier élément et on lui réapplique la fonction fd."
On est rapidement convaincu de ce que l'on a bien défini la fonction désirée.
Malheureusement, cette expression n'a pas de sens en LISP et le membre de droite de l'égalité ne suffit pas à définir fd, puisque la signification de l'identificateur "fd" n'y est pas précisée. Pour cette raison, on possède, en LISP, une règle particulière, permettant de lier un identificateur à une fonction de type λ : on définira fd par l'expression:

$$label[fd;\lambda[[x];[atom[cdr[x]]\rightarrow car[x];T\rightarrow fd[cdr[x]]]]]$$

fd : identificateur ; $\lambda[[x];[atom[cdr[x]]\rightarrow car[x];T\rightarrow fd[cdr[x]]]]$: définition de la fonction désignée par l'identificateur "fd"

symbole spécial, avertissant qu'on va trouver entre "[ ]" l'identificateur d'une fonction suivi de sa définition en terme de fonction-λ .
Une telle expression est appelée: <u>fonction-label.</u>

Exemple.

Cherchons le dernier élément de la liste (B,C) avec la fonction "fd". Le programme à écrire est le suivant:

label[fd;λ[[x];[atom[cdr[x]]→car[x];T→fd[cdr[x]]]][(B,C)]

(fonction-label, argument)

= λ[[x];[atom[cdr[x]]→car[x];T→fd[cdr[x]]]][(B,C)]

(fonction-λ associée, argument)

= [atom[cdr[(B,C)]]→car[(B,C)];T→fd[cdr[(B,C)]]]

(remplacement de x par sa valeur.)

= [F→B;T→fd[cdr[(B,C)]]]

= fd[cdr[(B,C)]]

= fd[(C)] (identificateur de fonction, argument)

= λ[[x];[atom[cdr[x]]→car[x];T→fd[cdr[x]]]][(C)]

(remplacement de l'identificateur par la fonction-λ associée.)

= [atom[cdr[(C)]]→car[(C)];T→fd[cdr[(C)]]]

= [T→car[(C)];T→fd[cdr[(C)]]]

= car[(C)]

= C .

§ 5. Arguments fonctionnels.

Nous n'avons considéré, jusqu'ici, que des fonctions ayant pour arguments des formes, ainsi dans l'exemple

car[cons[x;y]] ,

la fonction "car" a pour argument la forme cons[x;y] constituée, elle même, de l'application de la fonction "cons" aux formes élémentaires "x" et "y".

Il est permis, en LISP, de définir des fonctions, ayant pour arguments une ou plusieurs fonctions. Un exemple très simple de telle fonction est:

λ[[x;fn];fn[x]]

qui est la fonction associant à une variable et une fonction, l'application de la fonction à la variable.
On a:

- λ[[x;fn];fn[x]][(A.B);car] = car[(A.B)] = A
- λ[[x;fn];fn[x]][A;λ[[x];cons[x;NIL]]]
  = λ[[x];cons[x;NIL]][A] = cons[A;NIL] = (A) .

Le passage des arguments fonctionnels diffère, en LISP, radicalement du passage des arguments qui sont des formes. Considérons l'exemple suivant de programme LISP: (*)

λ[[x;y];λ[[x;fn];fn[x]][cons[x;y];λ[[x];cons[x;y]]]][A;NIL] (1)

= λ[[x;fn];fn[x]][cons[A;NIL];λ[[x];cons[x;NIL]]] (2)

= λ[[x;fn];fn[x]][(A);λ[[x];cons[x;NIL]]] (3)

= λ[[x];cons[x;NIL]][(A)] (4)

= cons[(A);NIL] = ((A)) (5)

La séquence (2), (3), (4) montre que l'argument cons[A;NIL] qui est une forme peut être évalué, avant d'être substitué dans le corps de fonction: fn[x] , tandis que l'argument fonctionnel λ[[x];cons[x;NIL]] doit être substitué tel quel.

(*) La suite du § 5 peut être omise en première lecture, car l'exemple traité peut paraître compliqué. On pourra y revenir après lecture du chapitre 4 décrivant la sémantique du LISP.

On dit que les arguments qui sont des formes, sont passés par valeur, tandis que les arguments qui sont des fonctions sont passés par nom.

La raison d'un tel comportement, c'est que des formes ne représentent, finalement, que des valeurs, pouvant être calculées, une fois pour toutes, avant d'être passées au corps de fonction, tandis que des fonctions décrivent un traitement à entreprendre et doivent donc être passées telles quelles.
Le passage des arguments fonctionnels en LISP, pose, en gros, les mêmes problèmes que le passage des procédures comme paramètres formels en Algol 60. La façon de les résoudre est, aussi, semblable.

Chapitre 3 : La syntaxe du LISP.

Le chapitre précédent constituait une description du LISP où l'on mélangeait les notions sémantiques et syntaxiques; dans ce chapitre, nous donnerons une définition rigoureuse de la syntaxe du langage décrit intuitivement, plus haut, et appelé M-langage. (*) Ensuite, nous donnerons un ensemble de règles, permettant de transformer tout programme, forme ou fonction LISP en S-expression. Le langage ainsi obtenu s'appelle S-langage et son intérêt est lié au fait que, jusqu'à présent, c'est uniquement ce langage qu'on utilise pour présenter des programmes LISP à une machine.

Il semble que la raison pour laquelle on n'utilise pas le M-langage pour le passage en machine soit qu'en S-langage, un programme LISP est une S-expression et peut donc être argument d'un autre programme, cet avantage, fort théorique, pourrait être utilisé pour modifier ou même créer un programme en cours d'exécution. Il est assez rare qu'on utilise cette possibilité. (**) A part cela, il n'y aurait aucune difficulté à établir un programme de traduction automatique du M-langage en S-langage et donc de permettre la programmation en M-langage.

§ 1. La syntaxe du M-langage.

- Une <u>forme</u> est

soit une <u>constante</u>, c'est à dire une S-expression,

soit une <u>variable</u>, c'est à dire <u>un identificateur</u> formé d'une suite de lettres minuscules et de chiffres, commençant par une lettre,

soit une <u>forme conditionnelle</u>, c'est à dire une expression du type $[\varepsilon_1 \Rightarrow \omega_1 ; \ldots ; \varepsilon_n \Rightarrow \omega_n]$ où les ε_i et les ω_i sont des formes et $n \geqslant 1$,

soit un <u>argument fonctionnel</u>, c'est à dire une fonction,

soit <u>l'application d'une fonction</u>, c'est à dire une expression du type $\varphi[\varepsilon_1 ; \ldots ; \varepsilon_n]$ où les ε_i sont des formes, φ est une fonction et $n \geqslant 0$.

(*) M-langage est une abréviation de "Méta-langage". En fait, le M-langage n'est pas différent, aux notations près, du S-langage. La raison de cette appellation est que, peu après la création du LISP, son auteur a donné une description complète et concise de sa sémantique entièrement écrite en M-langage. D'où le nom de "Méta-langage". Nous ne donnerons pas ici cette description, on la trouvera, notamment, dans les références [1], [2], [3].

(**) Nous en trouverons un exemple, très simple, dans le programme de simplification des fonctions algébriques de la cinquième partie.

- Une fonction est

soit un des algorithmes primitifs: "car" , "cdr" , "cons" , "atom" , "eq" ,

soit un identificateur,

soit une fonction-λ, c'est à dire une expression du type $\lambda[[x_1; \ldots ;x_n];\varepsilon]$ où ε est une forme, les x_i sont des identificateurs et $n \geq 0$,

soit une fonction-label, c'est à dire une expression du type label$[\psi;\theta]$ où ψ est un identificateur et θ est une fonction.

- Un programme LISP est

une application de fonction du type $\varphi[\alpha_1; \ldots ;\alpha_n]$ où les α_i sont des constantes.

§ 2. Le S- langage: règles de passage du M-langage au S-langage.

Convenons de noter ε ou ω une forme du M-langage et par ε^* ou ω^* sa transformée en S-expression; par φ , θ ou ψ une fonction du M-langage et par φ^* , θ^* ou ψ^* sa transformée en S-expression. Les règles de transformation sont les suivantes:

Règles pour les formes.

(ES 1) - Si ε est une constante α, ε^* est (QUØTE,α).

(ES 2) - Si ε est une variable, ε^* est le symbole atomique obtenu en remplaçant toutes les lettres minuscules de ε par les majuscules correspondantes.

(ES 3) - Si ε est une forme conditionnelle $[\varepsilon_1 \rightarrow \omega_1; \ldots ;\varepsilon_n \rightarrow \omega_n]$, ε^* est la liste (CØND($\varepsilon_1^*,\omega_1^*$) ... ($\varepsilon_n^*,\omega_n^*$)).

(ES 4) - Si ε est un argument fonctionnel φ , ε^* est (FUNCTIØN,φ^*).

(ES 5) - Si ε est l'application d'une fonction $\varphi[\varepsilon_1; \ldots ;\varepsilon_n]$, ε^* est $(\varphi^*,\varepsilon_1^*, \ldots ,\varepsilon_n^*)$.

Règles pour les fonctions.

(AS 1) - Si φ est un identificateur, règle identique à ES2.

(AS 2) - Si φ est une fonction-λ , $\lambda[[x_1; \ldots ;x_n];\varepsilon]$, φ^* est $(\text{LAMBDA}(x_1^*, \ldots ,x_n^*)\ \varepsilon^*)$.

(AS 3) - Si φ est une fonction-label, $\text{label}[\psi;\theta]$, φ^* est $(\text{LABEL},\psi^*,\theta^*)$.

Règle pour les programmes LISP. (PS I)

Puisqu'un programme LISP $\varphi[\alpha_1; \ldots ;\alpha_n]$ est un cas particulier de forme du type "application d'une fonction", sa transformée est normalement $(\varphi^*(\text{QUØTE},\alpha_1) \ldots (\text{QUØTE},\alpha_n))$. Nous conviendrons, cependant, de l'écrire, plus simplement, $(\varphi^*,\alpha_1, \ldots ,\alpha_n)$. Il est bon de remarquer que cela n'est possible que parce que l'on sait que, dans ce cas, toutes les formes α_i sont des constantes.

Exemples de transformations:

M-langage	S-langage
(A.(B.C))	(QUØTE(A.(B.C)))
$\lambda[[x];\text{cons}[x;A]]$	(LAMBDA(X)(CØNS,X(QUØTE,A)))
$[\text{atom}[x]\rightarrow x;T\rightarrow \text{car}[x]]$	(CØND((ATØM,X)X)((QUØTE,T)(CAR,X)))

Chapitre 4 : La sémantique du LISP.

§ I. Introduction.

Puisque nous connaissons, maintenant, la syntaxe exacte du LISP, nous pouvons écrire tous les programmes LISP possibles, cependant, leur signification ne nous est pas connue, si ce n'est, intuitivement, grâce au chapitre 2.

Le but de ce chapitre est de fixer sans ambiguïté la signification de tout programme LISP. Pour connaître cette signification, nous allons définir un "programme" nommé "evalquote" qui, lorsqu'on lui fournit comme donnée un programme LISP écrit en S-langage, a pour résultat une S-expression qui est, par définition, le résultat du programme LISP. En d'autres mots, "evalquote" interprète tout programme LISP.

Le programme "evalquote" se décompose, principalement, en deux sous-programmes " apply" et "eval" dont le rôle est:

- pour "apply" : application d'une fonction à une liste de constantes,

- pour "eval" : évaluation d'une forme.

Ces deux programmes se réappellent l'un l'autre très souvent et utilisent,en commun,une liste de paires pointées,nommée "a-liste",comportant des paires du type (<identificateur>. <S-expression>).Le rôle de cette liste est d'associer aux variables leur signification à un moment donné de l'évaluation .

Exemple de a-liste.

((X.(A.B)) (Y.(C.D)) (FN.(LAMBDA(X)X)))

Une telle a-liste signifie :

"X désigne la S-expression (A.B) ."

"Y désigne la S-expression (C.D) ."

"FN désigne la S-expression (LAMBDA(X)X) ."

§ 2. Le sous-programme "apply" :

application d'une fonction à une liste de constantes .

Ce sous-programme a 3 arguments :

- une fonction φ^* (*)
- une liste de constantes $(\alpha_1, \ldots, \alpha_n)$
- la a-liste : a .

"apply" a pour action d'appliquer la fonction φ^* à la liste d'arguments $(\alpha_1, \ldots, \alpha_n)$.
On écrit :

$$\varphi[\alpha_1; \ldots; \alpha_n] = \text{apply}[\varphi^*; (\alpha_1, \ldots, \alpha_n); \text{NIL}]$$

(par définition) ,

ce qui signifie : " la valeur de $\varphi[\alpha_1; \ldots; \alpha_n]$ est, par définition, le résultat de l'exécution du sous-programme "apply" avec les arguments φ^* , $(\alpha_1, \ldots, \alpha_n)$, NIL ".

Cas A1 : φ^* est l'un des symboles atomiques "CAR","CDR","CØNS", "EQ","ATOM".

Le résultat de "apply" est alors, respectivement, $\text{car}[\alpha_1]$, $\text{cdr}[\alpha_1]$, $\text{cons}[\alpha_1; \alpha_2]$, $\text{eq}[\alpha_1; \alpha_2]$, $\text{atom}[\alpha_1]$.

Cas A2 : φ^* est un autre symbole atomique.

Dans ce cas, on va chercher la définition de la fonction φ^* sur la a-liste : on examine toutes les paires pointées de la a-liste, en commençant par la gauche. Si on trouve une paire du type $(\varphi^* . \psi^*)$, "apply" se réappelle, lui-même, avec les trois arguments ψ^* , $(\alpha_1, \ldots, \alpha_n)$, a et on a :

$$\text{apply}[\varphi^*; (\alpha_1, \ldots, \alpha_n); a] = \text{apply}[\psi^*; (\alpha_1, \ldots, \alpha_n); a]$$

(par définition) ,

Si on ne trouve aucune paire pointée de ce type, "apply" délivre un message d'erreur et l'exécution s'arrête .

(*) Nous convenons de noter φ^* , ε^* , les fonctions et les formes, au cours de ce chapitre, pour rappeler qu'elles sont écrites sous forme de S-expressions .

Cas A3 : φ^* est une fonction-λ : $(\text{LAMBDA}(x_1^*, \ldots, x_n^*)\varepsilon^*)$.

Le rôle de "apply" , dans ce cas, est de préparer la a-liste pour l'évaluation de ε^* , par "eval" : "apply" commence par constituer une nouvelle a-liste en rajoutant les n paires pointées $(x_n^*.\alpha_n) \ldots (x_1^*.\alpha_1)$ au début de a.
Ensuite, "apply" passe la main au programme "eval" avec comme arguments ε^* et la nouvelle a-liste.
En désignant par a_{anc} la a-liste initiale et

par a_{nov} la a-liste transformée, on a :

$$\text{apply}[\varphi^*;(\alpha_1, \ldots, \alpha_n);a_{anc}] = \text{eval}[\varepsilon^*;a_{nov}]$$

(par définition) .

Cas A4 : φ^* est une fonction-label : $(\text{LABEL}, \psi^*, \theta^*)$.

On constitue une nouvelle a-liste, en rajoutant la paire pointée $(\psi^*.\theta^*)$ au début de l'ancienne, de façon à donner un sens à l'identificateur ψ^* durant la suite de l'exécution.
Normalement, θ^* est une fonction-λ (voir page I.16); "apply" va, alors, se réappeler lui-même avec les arguments θ^* , $(\alpha_1, \ldots, \alpha_n)$, a_{nov} et on a :

$$\text{apply}[\varphi^*;(\alpha_1, \ldots, \alpha_n);a_{anc}] = \text{apply}[\theta^*;(\alpha_1, \ldots, \alpha_n);a_{nov}]$$

(par définition) .

Lorsque, dans la suite, on rencontrera l'identificateur de fonction ψ^* (cas A2), on ira chercher sa définition sur la a-liste et la présence de la paire pointée $(\psi^*.\theta^*)$ nous avertira de ce que θ^* doit remplacer ψ^* pour la suite de l'exécution.

Cas A5 : φ^* est $(\text{FUNARG}, \psi^*, a')$.

Dans ce cas, a' est une liste de paires pointées semblable à la a-liste. Normalement, c'est, d'ailleurs, une valeur antérieure de la a-liste qu'on a stockée.
Ce cas nécessite une explication détaillée, car c'est la première fois qu'on rencontre le symbole "FUNARG" dans cet exposé.
Nous savons que lorsque une fonction ψ^* est argument d'une autre fonction, on doit écrire explicitement $(\text{FUNCTIØN}, \psi^*)$.
(règle ES 4 , page I.21)
C'est pour rappeler deux choses.

- Qu'il ne faut pas essayer d'évaluer ψ^*. (passage par nom)

- Qu'il faut garder la trace de l'environnement au moment de l'appel : si le corps de la fonction ψ^* contient

des variables libres, c'est à dire non liées par aucun symbole "λ" à l'intérieur de ψ^* , alors ces variables jouent le rôle de constantes pour cette fonction et il faut qu'à chaque emploi ultérieur de ψ^*, on leur redonne leur valeur au moment de l'appel. (*) C'est la raison pour laquelle l'argument (FUNCTIØN,ψ^*) est passé à "apply" sous la forme (FUNARG,ψ^*,a') où a'est la a-liste au moment de l'évaluation de (FUNCTIØN,ψ^*); a' permet de trouver les valeurs des variables libres de ψ^*.
"apply" doit donc, avant d'appliquer ψ^* à la liste $(\alpha_1, \ldots, \alpha_n)$ restaurer la a-liste qui contient les "bonnes valeurs" des variables libres. Finalement, le traitement est le suivant :
"apply" se réappelle, lui-même, avec les arguments ψ^* , $(\alpha_1, \ldots, \alpha_n)$ et a' .
On a :

$$apply[\varphi^*;(\alpha_1, \ldots, \alpha_n);a] = apply[\psi^*;(\alpha_1, \ldots, \alpha_n);a'].$$

(par définition)

§ 3. Le sous-programme "eval" : évaluation d'une forme .

Ce sous-programme a 2 arguments :

- une forme ε^*
- la a-liste : a .

"eval" calcule la forme obtenue à partir de ε^* en remplaçant toutes les variables libres de ε^* par leur valeur trouvée sur la a-liste.

Cas E1 : ε^* est une constante : (QUØTE,α) .

Dans ce cas, le résultat de "eval" est α.

Cas E2 : ε^* est une variable. (un atome)

Dans ce cas, le résultat de "eval" est la valeur de ε^* trouvée sur la a-liste : on parcourt la a-liste à partir de la gauche jusqu'à trouver une paire pointée du type $(\varepsilon^*.\alpha)$
Si une telle paire pointée existe, le résultat de "eval" est α, sinon "eval" délivre un message d'erreur et l'exécution s'arrête.

(*) Voir l'exemple (1) de la page I.18 .

Cas E3 : ε^* est une forme conditionelle : $(C\emptyset ND(\varepsilon_1^*,\omega_1^*)\ldots(\varepsilon_n^*,\omega_n^*))$.

Dans ce cas, "eval" se réappelle, lui-même, pour calculer les formes ε_i^* jusqu'à en trouver une dont la valeur soit différente de "F", soit ε_j^* cette forme, alors "eval" se réappelle encore une fois, lui-même, avec les arguments ω_j^* et a. On a :

$eval[\varepsilon^*;a] = eval[\omega_j^*;a]$ (par définition) .

Si toutes les formes ε_i^* ont pour valeur "F" , alors le résultat de "eval" est "NIL" . Ce cas ne doit, en principe, jamais se produire.

Cas E4 : ε^* est un argument fonctionnel : $(FUNCTI\emptyset N,\varphi^*)$.

Dans ce cas, le résultat de "eval" est la liste $(FUNARG,\varphi^*,a)$. La raison de ce traitement est exposée à la page I.25 , cas A5 .

Cas E5 : ε^* est l'application d'une fonction : $(\varphi^*,\varepsilon_1^*, \ldots ,\varepsilon_n^*)$.

Dans ce cas, "eval" se réappelle n fois, lui-même, pour évaluer les n formes $\varepsilon_1^*, \ldots ,\varepsilon_n^*$ de valeurs respectives $\alpha_1, \ldots ,\alpha_n$; ensuite, il appelle "apply" avec les arguments $\varphi^*, (\alpha_1, \ldots ,\alpha_n)$, a . On a :

$eval[\varepsilon^*;a] = apply[\varphi^*;(\alpha_1, \ldots ,\alpha_n);a]$ (par définition) .

Nous pouvons comparer ce mécanisme avec celui du passage des arguments à une procédure en Algol 60 , en considérant la notion de fonction comme l'équivalent de la notion de procédure. Nous voyons que les formes ε_i^* sont évaluées avant d'être passées à φ^* par l'intermédiaire de "apply". Il s'agit donc d'un passage par valeur.
Cependant, nous l'avons vu plus haut, il faut pouvoir passer par nom les arguments fonctionnels, c'est pour cette raison qu'on introduit la forme $(FUNCTI\emptyset N,\psi^*)$ dont l'évaluation consiste seulement à créer la liste $(FUNARG,\psi^*,a)$, ce qui réalise bien un passage par nom de ψ^* .

§ 4. Le programme "evalquote" :

sémantique d'un programme LISP .

Ce programme a un seul argument qui est un programme LISP : $(\varphi^*,\alpha_1, \ldots ,\alpha_n)$. Le résultat délivré par "evalquote" est une S-expression qui est, par définition, la "valeur" ou encore l'"interprétation sémantique" de $(\varphi^*, \ldots ,\alpha_n)$.
La seule action effectuée par "evalquote" est de passer la main à "apply", avec les arguments $\varphi^*,(\alpha_1, \ldots ,\alpha_n)$, NIL. On a donc:

$$evalquote[(\varphi^*,\alpha_1, \ldots ,\alpha_n)] = apply[\varphi^*;(\alpha_1, \ldots ,\alpha_n);NIL]$$
$$= \varphi[\alpha_1; \ldots ;\alpha_n] \text{ (par définition).}$$

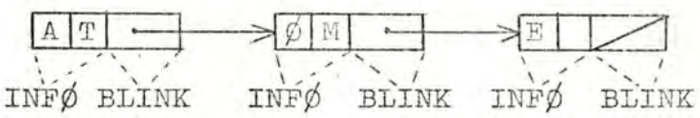
Chapitre 5 : Implémentation de l'Interpréteur pour le LISP "pur".

§ 1. Représentation interne des S-expressions.

Nous utiliserons des cellules de mémoire de 32 bits divisées en 2 champs de 16 bits. Ces cellules sont numérotées de 1 à 32.000. Leur numéro d'ordre est appelé "adresse". Lorsque le premier champ est supposé contenir une adresse, il est désigné par ALINK. Dans ce cas, le premier bit du champ est un bit de signe, les 15 autres bits donnent la valeur absolue de l'adresse. Ce champ peut aussi contenir deux caractères de huit bits, dans ces conditions, il est désigné par INFØ.
Le second champ est désigné par BLINK et doit toujours contenir une adresse. Sa configuration est semblable à celle de ALINK.

Représentation interne d'un atome.

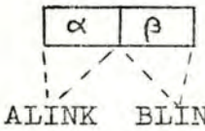
Considérons un atome formé de n caractères et soit m le plus grand nombre entier inférieur ou égal à (n+1)/2. Dans notre système, on utilise m cellules de mémoire pour représenter cet atome.
Dans sa partie INFØ, la première cellule contient les deux premiers caractères de la représentation externe de l'atome, la deuxième cellule contient les deux caractères suivants et ainsi de suite, jusqu'à la dernière cellule qui contient les deux derniers caractères du symbole, s'il possède un nombre pair de caractères et son dernier caractère suivi d'un blanc, sinon.
Dans sa partie BLINK, chaque cellule contient l'adresse de la cellule suivante, sauf la dernière qui contient ⊠ .
Par exemple, le symbole atomique "ATØME" donnera lieu à l'assemblage de cellules suivant.



C'est l'adresse de la première cellule qui représentera l'atome partout en mémoire.

Représentation interne d'une paire pointée.

Une paire pointée $(\alpha.\beta)$ est représentée en mémoire par l'adresse d'une cellule contenant en partie ALINK la représentation de α et en partie BLINK la représentation de β .
On schématise une telle cellule de la façon suivante:



(*)

(*) Nous désignerons toujours, dans la suite, par le même symbole la représentation externe et la représentation interne d'une S-expression, lorsque cela ne crée pas d'ambiguïté.

Il est évidemment capital de pouvoir tester l'atomicité d'une S-expression α . Or, il se fait que dans l'ordinateur utilisé pour réaliser notre système, les caractères ont toujours le bit de gauche égal à 1. Par conséquent, si une cellule contient deux caractères en INFØ, la configuration de bits de ce champ est celle d'une adresse négative. Par contre, si ALINK contient l'adresse d'une autre S-expression, la configuration de bits est celle d'une adresse positive. Finalement, on a le critère suivant:

α est l'adresse d'un symbole atomique ssi: ALINK(α)<0.

Ce test est évidemment lié à la machine utilisée.

§ 2. Organisation de la mémoire.

Une partie de la mémoire constitue une zone réservée. Elle est occupée par les représentations internes des fonctions de base: "CAR" , "CDR" , "CØNS" , "EQ" , "ATØM" et celles d'autres atomes jouant un rôle fondamental dans le fonctionnement de l'interpréteur: "CØND" , "F" , "FUNARG" , "FUNCTIØN" , "LABEL" , "LAMBDA" , "NIL" , "QUØTE" , "T" . De plus, elle contient encore des listes utilisées lors de la lecture et par le "garbage collector".
Le reste de la mémoire est organisé comme une liste linéaire appelée "liste libre" et qui contient, à un moment donné toutes les cellules disponibles. Un emplacement de mémoire appelé FREE contient un pointeur vers le début de cette liste.

Le choix d'une cellule disponible est dès lors très simple à réaliser: on utilise la cellule d'adresse FREE et on fait,

```
FREE:=BLINK(FREE);
```

Lorsqu'une cellule de mémoire utilisée est devenue inutile, c'est à dire lorsqu'elle n'est plus accessible par le programme LISP, elle n'est pas réintégrée immédiatement à la liste libre. C'est seulement lorsqu'on voudra extraire une cellule de la liste libre et que celle-ci sera vide que l'on appellera le programme "garbage collector" qui reconnaîtra toutes les cellules inaccessibles et constituera avec celles-ci une nouvelle liste libre. Le détail de ce fonctionnement est exposé dans la quatrième partie consacrée au "garbage collector".

Nous supposerons pour l'instant qu'il existe un sous-programme nommé "EXFREE" de trois arguments X ,Y ,Z. qui renvoie dans Z l'adresse d'une cellule disponible, cette cellule contient X en partie ALINK et Y en partie BLINK.

L'interpréteur est la réalisation de la fonction "evalquote", il se compose donc essentiellement de deux sous-programmes "apply" et "eval". Ces programmes sont récursifs et s'appellent mutuellement. Ils utilisent une pile-liste dont l'adresse de la première cellule est contenue dans le pointeur STACK.
Par pile-liste, on entend une liste qui se comporte comme un stack. On a la correspondance suivante:

- le stack est vide.	- STACK=NIL .
- mettre Y au sommet du stack.	- EXFREE(Y,STACK,STACK); .
- mettre dans Y la valeur au sommet du stack.	- Y:=ALINK(STACK); .
- supprimer un niveau du stack.	- STACK:=BLINK(STACK); .

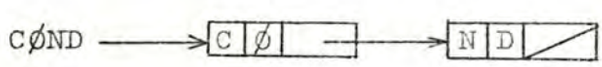
Au cours de l'exécution, lorsqu'on se branche récursivement à "eval" ou à "apply", on range dans cette pile-liste les résultats partiels à préserver, les listes nécessaires à la suite de l'exécution ainsi qu'un indicateur permettant de retrouver, par la suite, l'endroit où l'on a effectué le branchement.

Une position de mémoire particulière nommée ACCU, contient le résultat de chaque exécution de "eval" ou de "apply". Ce résultat sera la plupart du temps stocké dans la pile-liste pour être utilisé dans la suite de l'exécution. Il est très important de remarquer qu'un symbole atomique ne possède qu'une seule représentation interne, c'est à dire une seule adresse. La façon dont cela est réalisé par le programme de lecture sera exposée dans la troisième partie consacrée aux procédures d'entrée/sortie et au superviseur.

Signalons enfin, l'existence des pointeurs suivants.

- Le pointeur ALIST pointe vers la première cellule de la a-liste.

- A chaque symbole atomique de base: "CØND", "F" , ... ,"T" est associé un pointeur de même nom que le symbole, contenant sa représentation interne.



- Avant chaque branchement à "apply", le pointeur PHI contient la fonction à appliquer à la liste de constantes contenue dans le pointeur LIST.

- Avant chaque branchement à "eval", le pointeur EPSI pointe vers la forme à évaluer.

- Le pointeur Q∅ contient l'adresse de la représentation interne du programme LISP après lecture par le programme d'entrée.

- Les pointeurs P,Q,R, ... servent à stocker des résultats intermédiaires.

§ 3. Remarques sur la façon de présenter les algorithmes.

Dans la description de l'interpréteur, nous utiliserons un formalisme semblable à celui des langages tels que Algol 60. Nous supposons que toutes les règles et conventions utilisées sont évidentes. Nous donnons toutefois quelques précisions concernant des cas pouvant prêter à confusion.

- Un texte mis entre "[ ]" dans le texte d'un algorithme représente un commentaire.

- Une expression du type:

si <condition>, <instruction>
sinon <instruction>

a la même signification que

if <condition> then <instruction>
else <instruction> ,

en Algol.

- Le symbole réservé fin ne représente pas la fin d'un bloc, mais provoque l'arrêt d'un sous-programme et le retour au programme appelant.

- Une série d'instructions situées à droite d'une accolade est analogue à un "compound statement" en Algol 60.

§ 4. Description de l'implémentation de l'interpréteur.

Le programme "evalquote".

Un programme LISP, dans notre système, est une liste du type

$$(\varphi^*, \alpha_1, \ldots, \alpha_n)$$

où φ^* et les α_i sont des S-expressions. Après lecture, on a en mémoire la situation suivante.

Le travail de "evalquote" se réduit à passer la main à "apply" avec pour arguments la fonction φ^*, la liste de constantes $(\alpha_1, \ldots, \alpha_n)$ et une a-liste vide. Par conséquent, "evalquote" est le programme suivant:

```
sous-programme EVALQUØTE;
               imprimer "DEBUT DE EVALQUØTE";
               PHI:=ALINK(QØ);
               LIST:=BLINK(QØ);
               ALIST:=NIL;
               aller à Apply;

findeevalquote : imprimer "FIN DE EVALQUØTE";
               fin;
```

Le sous-programme "apply".

Les S-expressions traitées par ce sous-programme, lui sont passées par l'intermédiaire des trois pointeurs PHI , LIST , ALIST.

- PHI contient la représentation de la fonction φ^* ,

- LIST contient la représentation de la liste de constantes $(\alpha_1, \ldots, \alpha_n)$,

- ALIST contient l'adresse de la a-liste.

Phase de sélection.

Selon la nature de la fonction φ^*, le traitement effectué par "apply" variera. Par conséquent, la première chose à faire en entrant dans "apply", c'est de déterminer la nature de la fonction φ^*.

Il y a cinq cas possibles.

Si φ^* est une fonction primitive: "CAR" , "CDR" , "CØNS" , "ATØM" ou "EQ", alors, PHI contient une adresse comprise entre 1 et 5. Car, on a décidé d'organiser la mémoire de telle sorte que les représentations internes de "CAR" , ... , "EQ" soient les adresses 1 à 5.
De plus, chacune des fonctions primitives est réalisée dans le corps de l'interpréteur par un sous-programme précédé d'une étiquette du type " label(i): " où i = 1, ... , 5.
En conséquence, lors de la phase de sélection de "apply", on se branchera à label (PHI) .

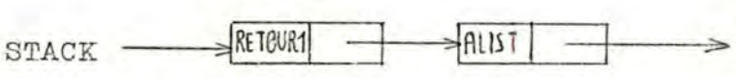
Si φ^* est un autre symbole atomique, on se branchera à apply2 . Ce cas est reconnu par le fait que ALINK(PHI)<0 .

Si aucun de ces deux cas n'est vérifié, φ^* est soit une fonction-λ, soit une fonction-label, soit un argument fonctionnel. Et dans ces trois cas, "apply" modifiera la a-liste au cours de son exécution. Cependant, les valeurs rajoutées ou supprimées par "apply" devront être restaurées en fin d'exécution de celui-ci.
Ce mécanisme est comparable à celui d'Algol 60 pour les blocs et les procédures: on peut établir la table de correspondance suivante.

Algol 60	LISP
- Entrée dans un bloc: création des variables locales.	- Entrée dans "apply" avec une fonction-λ ou une fonction-label: addition de paires pointées à la a-liste.
- Appel d'une procédure: utilisation du display correspondant à la déclaration de procédure.	- Entrée dans "apply" avec un argument fonctionnel: utilisation de la a-liste courante au moment de l'évaluation de l'argument fonctionnel.
- Sortie d'un bloc ou d'une procédure: restauration de l'environnement de l'entrée ou de l'appel.	- Fin de "apply": restauration de la a-liste courante du début de "apply".

Pour pouvoir restaurer la a-liste initiale à la fin de "apply", on la mémorisera en mettant sa valeur actuelle dans la pile-liste et un indicateur "RETØUR1" qui permettra de reconnaître la fin de "apply".

On aura donc la situation suivante:



Après avoir mis à jour la pile-liste, on pourra se brancher à apply3 , apply4 , apply5 suivant que φ^* est une fonction-λ, une fonction-label ou un argument fonctionnel.

Finalement, la phase de sélection de "apply" est réalisée par l'algorithme suivant:

```
Apply :     si PHI≤5, aller à label(PHI); [fonction primitive]
            si ALINK(PHI)<0, aller à apply2; [atome]
            EXFREE(ALIST,STACK,STACK); [mise à jour de la
                                        pile-liste.]
            EXFREE(RETØUR1,STACK,STACK);
            si ALINK(PHI)=LAMBDA, aller à apply3; [fonction-λ]
            si ALINK(PHI)=LABEL, aller à apply4; [fonction-label
            si ALINK(PHI)=FUNARG, aller à apply5; [argument
                                                   fonctionnel]
            imprimer "ERREUR AU DEBUT DE APPLY"; [erreur]
            ACCU:=PHI; aller à findeevalquote;
```

Voici maintenant la description du traitement des cinq cas précités.

Cas A1 : φ^* est une fonction primitive.

Dans ce cas, on applique l'opérateur primitif soit au premier, soit aux deux premiers éléments de la liste LIST et le résultat du calcul est mis dans le pointeur ACCU.
Si la liste contient plus d'éléments que l'opérateur n'en traite, les autres éléments sont ignorés mais il n'en résulte aucun message d'erreur.
Si la liste contient trop peu d'éléments, on obtiendra un résultat indéterminé qui risque de provoquer la détérioration de l'interpréteur lui-même.
Voici les cinq algorithmes correspondant aux cinq fonctions primitives.

```
label(1) :  ACCU:=ALINK(LIST); ["atom"]
            si ALINK(ACCU)<0, ACCU:=T;
            sinon ACCU:=F;
            aller à fin;

label(2) :  ACCU:=ALINK(LIST); ["car"]
            ACCU:=ALINK(ACCU);
            aller à fin;

label(3) :  ACCU:=ALINK(LIST); ["cdr"]
            ACCU:=BLINK(ACCU);
            aller à fin;

label(4) :  ACCU:=BLINK(LIST); ["cons"]
            EXFREE(ALINK(LIST),ALINK(ACCU),ACCU);
            aller à fin;
```

```
label(5) :      ACCU:=BLINK(LIST); ["eq"]
                si ALINK(LIST)=ALINK(ACCU), ACCU:=T;
                sinon ACCU:=F;
                aller à fin;
```

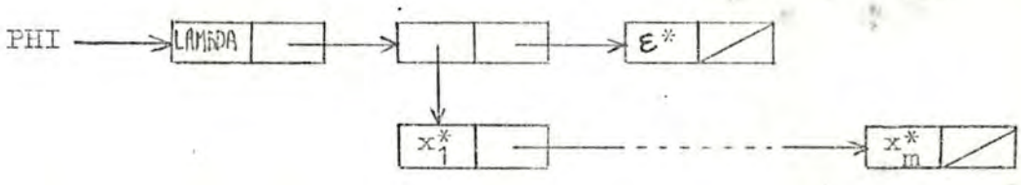
Cas A2 : φ^* est un autre symbole atomique.

Dans ce cas, on doit chercher sur la a-liste la définition de φ^*; si cette définition n'existe pas, il y a arrêt du programme "evalquote" tout entier, sinon on retourne au début de "apply" après avoir mis dans PHI la définition de φ^* :

```
apply2 :        ACCU:=ALIST;
apply21 :       si ACCU=NIL, {ACCU:=PHI; [erreur]
                              imprimer "FØNCTIØN NØN DEFINIE";
                              aller à findeevalquote;
                Q:=ALINK(ACCU);
                si ALINK(Q)=PHI, {PHI:=BLINK(Q);
                                  aller à Apply;
                ACCU:=BLINK(ACCU);
                aller à apply21;
```

Cas A3 : φ^* est une fonction-λ.

Dans ce cas, on a la situation suivante dans la mémoire.



Normalement, les x_i^* sont des variables figurant librement dans la forme ε^*, c'est pourquoi, on va donner une valeur à ces variables de façon à permettre l'évaluation de ε^* par "eval", en ajoutant au début de la a-liste les paires pointées

$$(x_1^* . \alpha_1) \ , \ \ldots \ , \ (x^*_{\min[n;m]} . \alpha_{\min[n;m]}) \ .$$

On a habituellement n=m, toutefois, si les deux listes sont de longueurs différentes, on constitue une liste de paires pointées de longueur égale à celle de la plus courte, qu'on concatène avec l'ancienne a-liste.
On aurait pu proscrire une telle éventualité et déclencher l'arrêt de "evalquote". L'optique utilisée ici est "optimiste": on essaie de continuer l'exécution le plus longtemps possible.

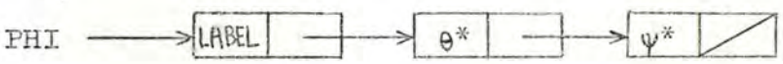
Après mise à jour de la a-liste, on se branche à "eval" pour évaluer la forme ε^*.

L'algorithme est:

```
apply3 :    P:=LIST;  [P contient la liste des constantes.]
            R:=BLINK(PHI);
            EPSI:=BLINK(R);
            EPSI:=ALINK(EPSI); [EPSI contient ε* .]
            R:=ALINK(R); [R contient la liste des variables.]
apply31 :   si R=NIL ou P=NIL, aller à Eval;
            EXFREE(NIL,ALIST,ALIST); [mise à jour a-liste.]
            EXFREE(ALINK(R),ALINK(P),ALINK(ALIST));
            P:=BLINK(P); R:=BLINK(R);
            aller à apply31;
```

Cas A4 : φ^* est une fonction-label.

Dans ce cas, on a la situation suivante, en mémoire.

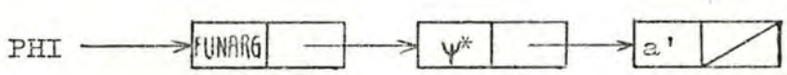


ψ^* est la fonction de définition de φ^* et de θ^*. On va donc mettre la paire pointée $(\theta^*.\psi^*)$ au début de la a-liste; ensuite, on retourne au début de "apply" après avoir mis dans PHI la définition de φ^*, c'est à dire ψ^*:

```
apply4 :    ACCU:=BLINK(PHI);
            PHI:=BLINK(ACCU);
            PHI:=ALINK(PHI); [PHI contient ψ* .]
            EXFREE(NIL,ALIST,ALIST); [mise à jour a-liste.]
            EXFREE(ALINK(ACCU),PHI,ALINK(ALIST));
            aller à Apply;
```

Cas A5 : φ^* est un argument fonctionnel.

Dans ce cas, on a la situation suivante:



ψ^* est une fonction et a' est la valeur de la a-liste au moment de l'évaluation de l'argument fonctionnel (FUNCTIØN,ψ^*). On met l'ancienne a-liste a' dans ALIST et ψ^* dans PHI; ensuite, on se branche au début de "apply". Dès lors, le calcul de $(\psi^*,\alpha_1, \ldots ,\alpha_n)$ se fera avec les valeurs des variables libres de ψ^* accessibles par la a-liste au moment de l'évaluation de (FUNCTIØN,ψ^*).

On trouvera, au chapitre 6, un exemple détaillé montrant l'évolution de la a-liste pour un programme LISP contenant des arguments fonctionnels.
L'algorithme est le suivant:

```
apply5 :    P:=BLINK(PHI);
            PHI:=ALINK(P); [PHI contient φ*.]
            P:=BLINK(P);
            ALIST:=ALINK(P); [ALIST contient a'.]
            aller à Apply;
```

Le sous-programme "eval".

"eval" évalue la forme dont la représentation est contenue dans le pointeur EPSI en se servant de la liste de paires pointées accessible par le pointeur ALIST.

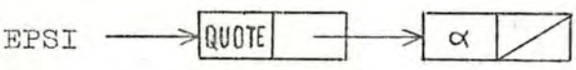
Phase de sélection.

Comme "apply" , "eval" débute par une phase de sélection: selon que la forme à évaluer est une constante, une variable, une forme conditionnelle, un argument fonctionnel ou l'application d'une fonction, on choisit le traitement adéquat:

```
Eval:       ACCU:=ALINK(EPSI);
            si ACCU<0, aller à  eval2; [variable]
            si ACCU=QUØTE, aller à eval1; [constante]
            si ACCU=CØND, aller à eval3; [forme conditionnelle]
            si ACCU=FUNCTIØN, aller à eval4;[argument
                                              fonctionnel]
            aller à eval5; [application d'une fonction]
```

Cas E1 : ε* est une constante.

Dans ce cas, on a en mémoire la situation suivante:



On met dans le pointeur ACCU, la valeur α de la constante:

```
eval1 :     ACCU:=BLINK(EPSI);
            ACCU:= ALINK(ACCU);
            aller à fin;
```

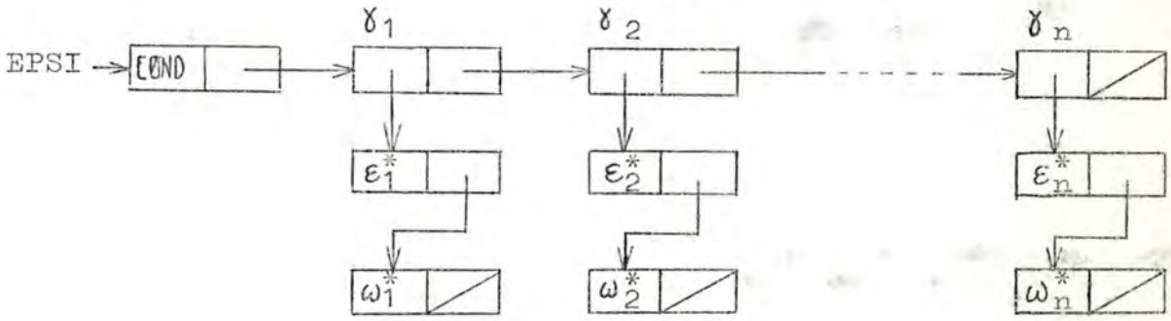
Cas E2 : ε^* est une variable.

Dans ce cas, on cherche la valeur de la variable en parcourant la a-liste de la gauche vers la droite. Si on trouve une paire pointée du type $(\varepsilon^*.\alpha)$, on met α dans le pointeur ACCU, sinon, il y a une erreur et "evalquote" se termine:

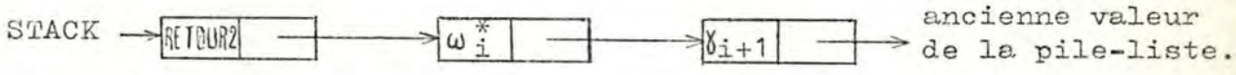
```
eval2 :     ACCU:=ALIST;
eval21 :    si ACCU=NIL, {ACCU:=EPSI; [erreur]
                          imprimer "VARIABLE SANS VALEUR";
                          aller à findeevalquote;
            Q:=ALINK(ACCU);
            si ALINK(Q)=EPSI, {ACCU:=BLINK(Q);
                               aller à fin;
            ACCU:=BLINK(ACCU);
            aller à eval21;
```

Cas E3 : ε^* est une forme conditionnelle.

Dans ce cas, on a la situation suivante:



La première action que "eval" doit effectuer, c'est de calculer les formes ε_i^* jusqu'à en trouver une dont la valeur n'est pas "F".
Le calcul d'une forme ε_i^* s'effectue par un branchement au début de "eval" précédé de la mise dans EPSI de ε_i^*.
Mais, auparavant, il faut avoir mémorisé ω_i^* et γ_{i+1}, pour pouvoir poursuivre l'exécution après évaluation de ε_i^*.
Cette mémorisation est réalisée en rajoutant trois cellules à la pile-liste comme indiqué ci-dessous.



Après évaluation de ε_i^*, on a la valeur de cette forme dans le pointeur ACCU et on poursuit le traitement de la façon suivante:

Si ACCU = F , on examine la nature de γ_{i+1} trouvé dans la pile-liste.

Si γ_{i+1} = NIL , le résultat final est "NIL".
Sinon on remplace dans la pile-liste

ω_i^* par ω_{i+1}^* ,

γ_{i+1}^* par γ_{i+2}^* ,

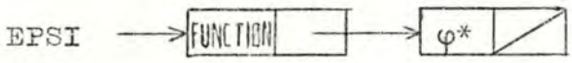
enfin, on va au début de "eval" pour calculer ε_{i+1}^*.

Si ACCU ≠ F, on supprime les trois premières cellules de la pile-liste, puis, on retourne au début de "eval" pour calculer la valeur de ω_i^* qui est aussi celle de ε^*.
Voici l'algorithme qui réalise cela:

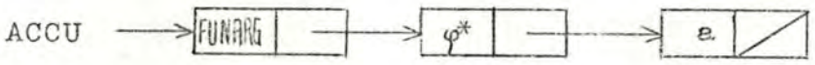
```
eval3 :     [mise à jour de la pile-liste et évaluation de ε*i.]
            EPSI:=BLINK(EPSI);
            EXFREE(BLINK(EPSI),STACK,STACK); [mettre γ2 dans
                                              la pile-liste.]
            ACCU:=ALINK(EPSI);
            EPSI:=ALINK(ACCU); [EPSI contient ε*1.]
            ACCU:=BLINK(ACCU);
            EXFREE(ALINK(ACCU),STACK,STACK); [mettre ω*i dans
                                              la pile-liste.]
            EXFREE(RETØUR2,STACK,STACK);
            aller à Eval;

retour2 :   [poursuite du traitement, après calcul de ε*i.]
            si ACCU≠F, ⎧ACCU:=BLINK(STACK);
                       ⎪EPSI:=ALINK(ACCU); [EPSI contient ω*i.]
                       ⎪STACK:=BLINK(ACCU);
                       ⎪STACK:=BLINK(STACK); [restauration de
                       ⎪                      la pile-liste.]
                       ⎩aller à Eval;
            P:=BLINK(STACK);
            EPSI:=BLINK(P);
            ACCU:=ALINK(EPSI);
            si ACCU=NIL, ⎧STACK:=BLINK(EPSI);
                         ⎩aller à fin;[tous les ε*i valent "F".]
            ALINK(EPSI):=BLINK(ACCU); [ mettre γi+2 dans la
                                        pile-liste.]
            ACCU:=ALINK(ACCU);
            Q:=BLINK(ACCU);
            ALINK(P):=ALINK(Q); [mettre ω*i+1 dans la pile-liste]
            EPSI:=ALINK(ACCU); [EPSI contient ε*i+1.]
            aller à Eval;
```

Cas E4 : ε^* est un argument fonctionnel.

Dans ce cas, on a la situation suivante.



Et le résultat de "eval" est:

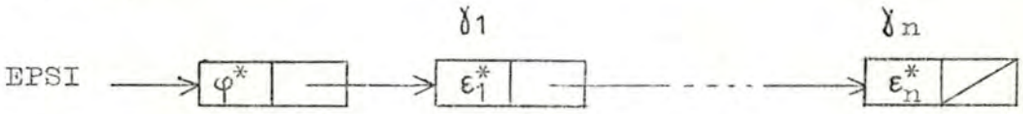


L'algorithme est:

```
eval4 :     EPSI:=BLINK(EPSI);
            EXFREE(ALIST,NIL,ACCU);
            EXFREE(ALINK(EPSI),ACCU,ACCU);
            EXFREE(FUNARG,ACCU,ACCU);
            aller à fin;
```

Cas E5 : ε^* est l'application d'une fonction.

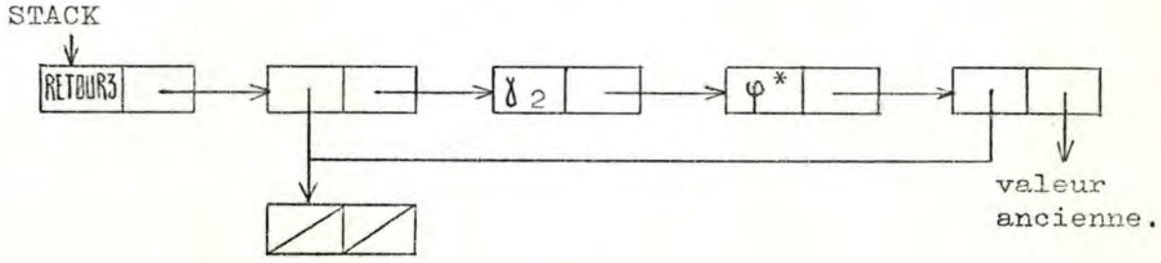
On a, dans ce cas, la situation suivante, en mémoire:



Le rôle de "eval" est, ici, d'évaluer les n formes ε_i^* en vue de passer la main à "apply" pour appliquer la fonction φ^* aux valeurs de ces n formes."eval" prépare cette évaluation en effectuant les opérations suivantes sur la pile-liste:

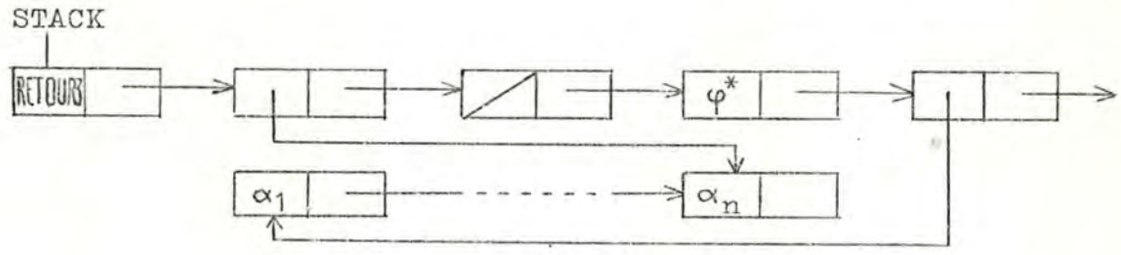
- rangement de φ^* ,
- rangement de la liste $(\varepsilon_2^*, \ldots, \varepsilon_n^*)$,
- préparation de la pile-liste pour la mémorisation des valeurs des formes ε_i^*,
- positionnement de l'indicateur de retour "RETØUR3".

Toutefois, si φ^* est une fonction sans argument, la pile-liste n'est pas modifiée et on va directement au début de "apply" avec φ^* et une liste de valeurs vide. Si ce n'est pas le cas, la mise à jour de la pile-liste a lieu et produit la situation ci-dessous.



Après cela, on va au début de "eval" pour calculer la valeur de ε_1^* .

Lorsqu'on a terminé d'évaluer une forme ε_i^*, dont la valeur se trouve dans ACCU, on range cette valeur dans la liste des constantes; si $i \neq n$, on retourne au début de "eval" pour calculer la valeur de ε_{i+1}^* , sinon on a la situation qui suit:



où les α_i sont les valeurs des ε_i^* . Il ne reste plus alors à "eval" qu'à passer la main à "apply" avec les arguments φ^* et $(\alpha_1, \ldots, \alpha_n)$ après avoir amputé la pile-liste de ses cinq premières cellules.
Voici l'algorithme qui réalise ces opérations:

```
eval5 :     LIST:=BLINK(EPSI);
            PHI:=ALINK(EPSI);
            si LIST=NIL, aller à Apply; [fonction
                                          sans argument.]
            EXFREE(NIL,NIL,ACCU); [mise à jour
                                     de la pile-liste.]
            EXFREE(ACCU,STACK,STACK);
            EXFREE(PHI,STACK,STACK); [rangement de φ*.]
            EXFREE(BLINK(LIST),STACK,STACK); [rangement
                                                de γ2 .]
            EXFREE(ACCU,STACK,STACK);
            EXFREE(RETØUR3,STACK,STACK);
            EPSI:=ALINK(LIST); [EPSI contient ε1*.]
            aller à Eval;

retour3 :   P:=BLINK(STACK);
            Q:=ALINK(STACK);
            ALINK(Q):=ACCU; [ranger αi dans la liste
                                       des constantes.]
            ACCU:=BLINK(P);
            R:=ALINK(ACCU);
            si R=NIL, ⎧PHI:=BLINK(ACCU); [tous les εi*
                      ⎪                    sont évalués.]
                      ⎪LIST:=BLINK(PHI);
                      ⎪STACK:=BLINK(LIST); [restauration
                      ⎪                      de la pile-liste ]
                      ⎪PHI:=ALINK(PHI); [PHI contient φ*.]
                      ⎪LIST:=ALINK(LIST); [LIST contient
                      ⎪                    (α1, ... ,αn).]
                      ⎩aller à Apply;
```

```
ALINK(ACCU):=BLINK(R); [évaluation de ε*i+1.]
EPSI:=ALINK(R); [EPSI contient ε*i+1.]
EXFREE(NIL,NIL,ACCU); [rajouter une cellule
                       à la liste des constantes.]
BLINK(Q):=ACCU;
ALINK(P):=ACCU;
aller à Eval;
```

Gestion de la pile-liste, commune à "eval" et "apply". (GS)

Lorsqu'on a fini d'évaluer une forme ou l'application d'une fonction à une liste de constantes, on se branche à l'étiquette "fin". A ce moment, le pointeur ACCU contient un résultat partiel; la suite du traitement dépend de la première cellule de la pile-liste.

Si elle contient l'indicateur "RETØUR1", ACCU contient le résultat de l'application d'une fonction-λ, d'une fonction-label ou d'un argument fonctionnel à une liste de constantes. Il faut restaurer la a-liste et la pile-liste telles qu'elles étaient avant ce traitement.

Si elle contient l'indicateur "RETØUR2", ACCU contient la valeur d'une forme prédicative qui est le premier élément d'un doublet dans une forme conditionnelle; la suite du traitement s'effectuera par branchement à l'étiquette retour2.

Si elle contient l'indicateur "RETØUR3", ACCU contient la valeur d'une forme ε^*_i argument d'une forme du type "application de fonction"; le traitement se poursuivra par l'exécution des instructions qui suivent l'étiquette retour3.

Si la pile-liste est vide, ACCU contient le résultat final de "evalquote".

On a donc l'algorithme suivant:

```
fin :   si STACK=NIL, aller à findeevalquote;
                      [retour au programme superviseur,
                       le résultat final est dans ACCU.]
        Q:=ALINK(STACK);
        si Q=RETØUR1, ⎧STACK:=BLINK(STACK);
                      ⎪ALIST:=ALINK(STACK); [restauration
                      ⎪                      a-liste.]
                      ⎪STACK:=BLINK(STACK); [restauration
                      ⎪                      pile-liste.]
                      ⎩aller à fin;
        si Q=RETØUR2, aller à retour2;
        si Q=RETØUR3, aller à retour3;
```

Chapitre 6 : Deux exemples.

§ 1. Premier exemple: Evolution de la pile-liste.

Nous allons maintenant décrire l'exécution d'un programme LISP très simple. Il s'agit du programme suivant:

label[ff;λ[[x];[atom[x]→x;T→ff[car[x]]]]][((A.B).C)] ,

qui s'écrit en S-langage :

```
((LABEL,FF(LAMBDA(X)(CØND((ATØM,X)X)
                          ((QUØTE,T)(FF(CAR,X))))))
 ((A.B).C))  .
```

La fonction "ff" appliquée à une S-expression a pour valeur le symbole atomique figurant le plus à gauche dans la représentation externe de la S-expression. Le résultat du programme ci-dessus est donc "A".

On a dessiné à la figure FIG 1

- la représentation du programme dans la mémoire telle qu'elle est constituée par le programme de lecture,

- les cellules extraites de la liste libre, au cours de l'évaluation du programme, pour construire la a-liste,

- les cellules extraites pour la constitution de la pile-liste.

A chaque cellule est associé un numéro qui représente son adresse et on a utilisé les deux conventions suivantes:

- les adresses des cellules de la a-liste et de la pile-liste suivent l'ordre chronologique d'extraction de la liste libre ,

- deux cases concaténées représentent une même cellule contenant des valeurs distinctes à des moments différents.

Le tableau T.1 décrit complètement l'exécution du programme.

- Les éléments de la colonne "Phase" donnent les différents cas de "apply" et de "eval" qui se déroulent successivement. Le chiffre accolé à la phase permet de distinguer différentes utilisations d'une même phase.

Par exemple, " E5 7 " signifie que c'est le cas E5 de "eval" qui se déroule et que ce cas a déjà été initialisé sept fois. On peut avoir plusieurs fois le même sigle, si le cas considéré appelle d'autres cas avant de pouvoir continuer son exécution.

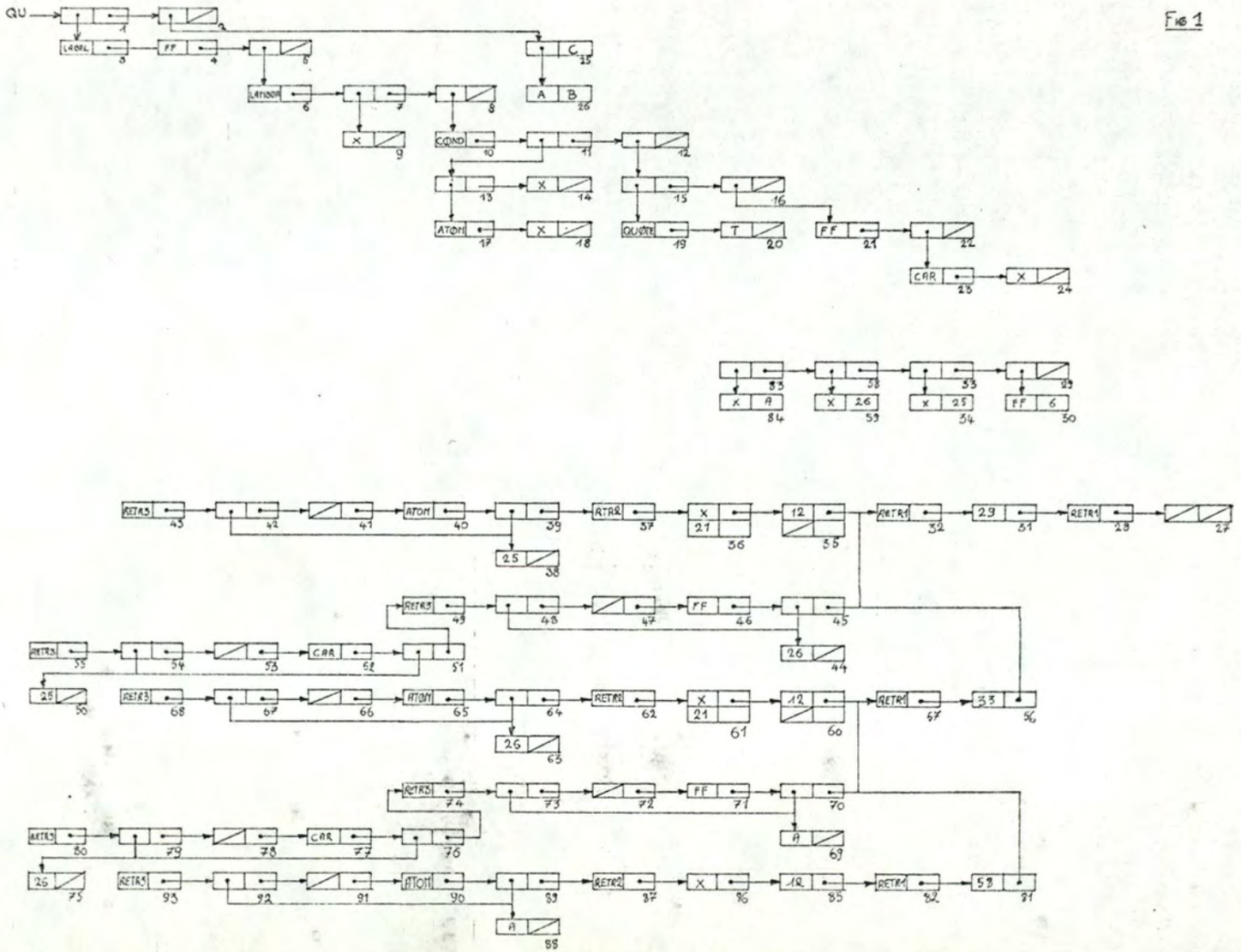
- En regard de la phase, on a indiqué les valeurs des pointeurs ACCU , LIST , PHI , EPSI , ALIST , STACK , au moment où elle passe la main à la phase suivante. Si une valeur n'est pas indiquée, cela signifie que le contenu du pointeur n'est pas utilisé par la phase suivante et est donc sans importance. La connaissance d'une ligne du tableau permet de calculer toutes les suivantes, c'est d'ailleurs de cette façon que travaille l'interpréteur.

Pour terminer, nous faisons la remarque suivante: chaque fois qu'on évalue une forme du type E5, la deuxième cellule et la cinquième cellule de la pile-liste contiennent un pointeur vers la même cellule, il semblerait qu'il y ait là une redondance. La raison c'est que toutes les fonctions utilisées ici n'ont qu'un seul argument; en général, ce nombre est supérieur à un et le premier pointeur permet d'atteindre la dernière cellule de la liste des valeurs, le second permet d'atteindre la première cellule de la liste.

Phase	ACCU	LIST	PHI	EPSI	ALIST	STACK
A4 1	4	2	6	NIL	29	28
A3 1	4	2	6	10	33	32
E3 1	14	2	6	17	33	37
E5 1				X	33	43
E2 1	25				33	43
E5 1		38	ATØM		33	37
A1 1	F				33	37
E3 1				19	33	37
E1 1	T				33	37
E3 1				21	33	32
E5 2				23	33	49
E5 3				X	33	55
E2 2	25				33	55
E5 3		50	CAR		33	49
A1 2	26				33	49
E5 2		44	FF		33	32
A2 1		44	6		33	32
A3 2				10	58	57
E3 2				17	58	62
E5 4				X	58	68
E2 3	26				58	68
E5 4		63	ATØM		58	62
A1 3	F				58	62
E3 2				19	58	62
E1 2	T				58	62
E3 2				21	58	57
E5 5				23	58	74
E5 6				X	58	80
E2 4	26				58	80
E5 6		75	CAR		58	74
A1 4	A				58	74
E5 5		69	FF		58	57
A2 2		69	6		58	57
A3 3				10	83	82
E3 3				17	83	87
E5 7				X	83	93
E2 5	A				83	93
E5 7		88	ATØM		83	87
A1 5	T				83	87
E3 3				X	83	82
E2 6	A				83	82
GS 1	A				58	57
GS 2	A				33	32
GS 3	A				29	28
GS 4	A				NIL	NIL

tableau T.1

FIG 1


QU
LABEL
FF
LABDOR
A
B
X
COND
ATOM
QUOTE
T
CAR
RETR3
RETR2
RETR1

§ 2. Second exemple: Evolution de la a-liste et passage des arguments fonctionnels.

Considérons le programme LISP suivant:

```
label[θ; λ[[x;y];
          [atom[y]→[eq[y;NIL]→NIL;
                    T→label[φ; λ[[y;fn];
                                 [atom[y]→NIL;
                                  T→cons[fn[car[y]];φ[cdr[y];fn]]]]]
                                [x; λ[[x];cons[x;y]]]];
          T→cons[θ[x;car[y]];θ[x;cdr[y]]]]]]
[(A,B);(C,D)]  .        (1)
```

Ce programme, dans lequel figure une fonction qui admet un argument fonctionnel : $\lambda[[x];cons[x;y]]$, combine les listes (A,B) et (C,D) pour fournir le résultat (((A.C)(B.C))((A.D)(B.D))). Remarquons que l'argument fonctionnel contient une variable libre et c'est ce qui fait l'intérêt de cet exemple.

L'exécution de (1) a lieu de la manière suivante:

$\theta[(A,B);(C,D)] = cons[\theta[(A,B);C];\theta[(A,B);(D)]]$

1. $\theta[(A,B);C] = \varphi[(A,B);\lambda[[x];cons[x;y]]]$ (y = C)
   $= cons[fn[car[y]];\varphi[cdr[y];fn]]$ (y = (A,B))

   1.1. $fn[A] = \lambda[[x];cons[x;y]][A]$ (y = C)
   $= (A.C)$

   1.2. $\varphi[(B);fn] = cons[fn[car[y]];\varphi[cdr[y];fn]]$ (y = (B))

      12.1. $fn[B] = (B.C)$ (y = C)

      12.2. $\varphi[NIL;fn] = NIL$

      12.3. $\varphi[(B);fn] = ((B.C))$

   1.3. $\theta[(A,B);C] = ((A.C)(B.C))$

2. $\theta[(A,B);(D)] = cons[\theta[(A,B);D];\theta[(A,B);NIL]]$

   2.1. $\theta[(A,B);D] = \varphi[(A,B);\lambda[[x];cons[x;y]]]$ (y = D)
   $= ((A.D)(B.D))$ (comme 1 .)

   2.2. $\theta[(A,B);NIL] = NIL$

   2.3. $\theta[(A,B);(D)] = (((A.D)(B.D)))$

3. $\theta[(A,B);(C,D)] = (((A.C)(B.C))((A.D)(B.D)))$.

Si on examine la façon dont le programme est exécuté, on voit, par exemple à la ligne 1.1. , que lors du calcul de fn[A] , la forme cons[x;y] n'est pas évaluée avec la valeur de y mise en dernier lieu sur la a-liste, c'est à dire (A,B), mais avec la valeur de y existant au moment du passage de l'argument fonctionnel λ[[x];cons[x;y]], c'est à dire "C".

Nous allons montrer que c'est bien ce traitement qui est effectué par l'interpréteur. Pour cela, nous réécrivons le programme (1) en S-langage :

```
((LABEL,THETA(LAMBDA(X,Y)(CØND
 ((ATØM,Y)(CØND((EQ,Y(QUØTE,NIL))(QUØTE,NIL))
                ((QUØTE,T)((LABEL,PHI(LAMBDA(Y,FN)(CØND
                                   ((ATØM,Y)(QUØTE,NIL))
                                   ((QUØTE,T)(CØNS(FN(CAR,Y))
                                                  (PHI(CDR,Y)
                                                      (FUNCTIØN,FN)))))))
                                  X(FUNCTIØN(LAMBDA(X)(CØNS,X,Y)))))))
 ((QUØTE,T)(CØNS(THETA,X(CAR,Y))(THETA,X(CDR,Y)))))))
 (A,B)(C,D))    .        (2)
```

La figure FIG.2 montre la représentation en mémoire de (2), ainsi que la totalité des cellules utilisées par l'interpréteur pour l'évolution de la a-liste. On a utilisé les mêmes conventions que dans l'exemple précédent. La pile-liste n'a pas été dessinée.

Considérons alors l'évaluation de fn[B] conformément à la ligne 12.1. A ce moment, on a la situation suivante:

PHI = FN , ALIST = 138 , LIST = (B) ;

on va alors chercher la définition de "FN" sur la a-liste, comme "FN" est un argument fonctionnel, on retournera au début de "apply" avec:

PHI = FN , ALIST = 128 , LIST = (B) ;

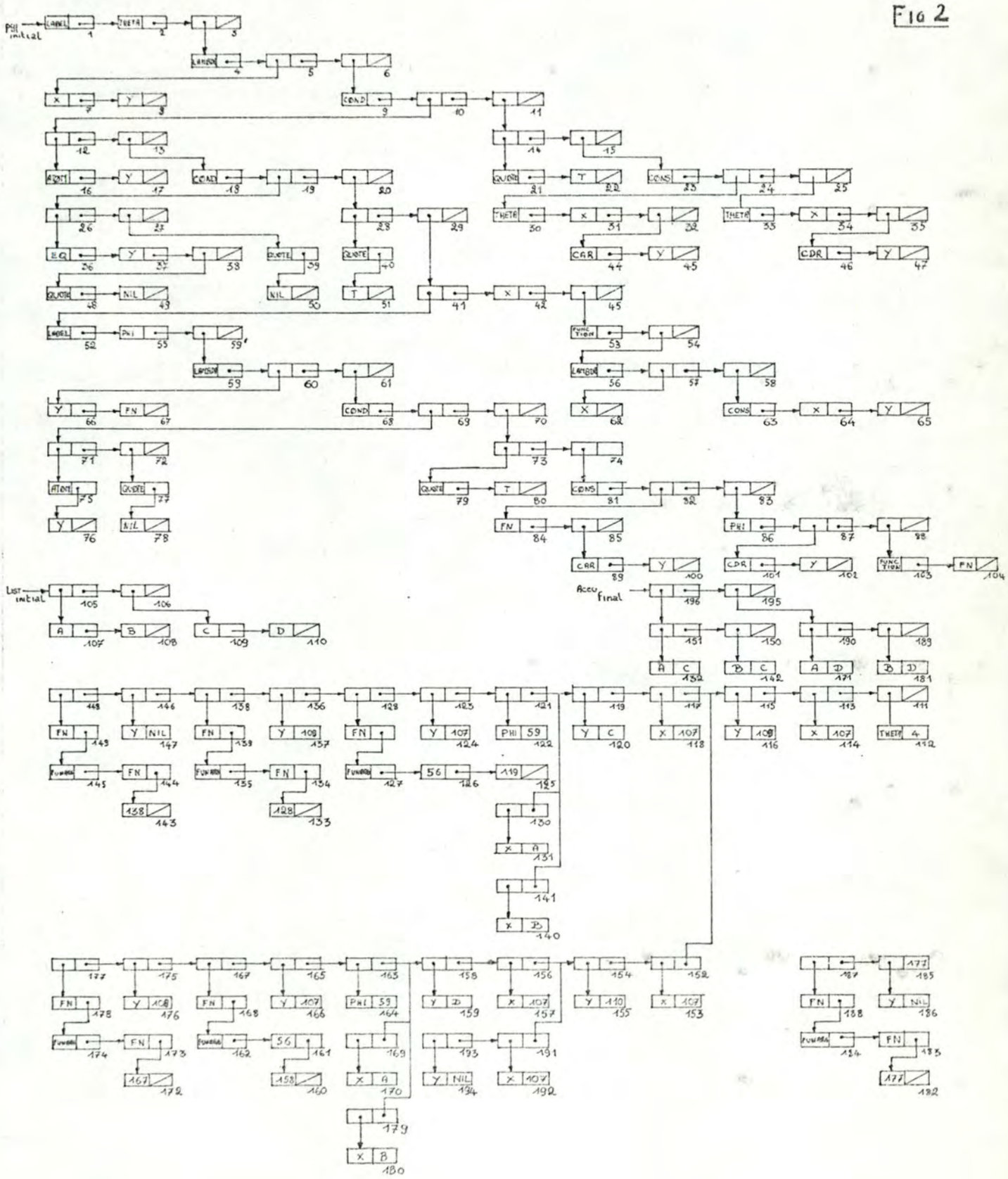
une nouvelle recherche de la valeur de "FN" provoque encore un branchement au début de "apply" avec:

PHI = 56 , ALIST = 119 , LIST = (B) .

Maintenant, la première valeur de "Y" accessible via la a-liste est bien "C" et l'exécution de "apply" produira le résultat annoncé.

On peut faire la remarque suivante:
Si dans la définition récursive de la fonction "PHI" , on avait écrit, au lieu de (FUNCTIØN,FN) , simplement FN, toutes les paires pointées de définition de "FN" dues à un appel récursif de "PHI", auraient été identiques à la première paire pointée de définition de "FN" et l'exécution aurait été accélérée.En écrivant (FUNCTIØN,FN) , on doit explorer la a-liste autant de fois qu'il y a eu d'appel récursif de "PHI", pour trouver la première définition de "FN".

FIG 2


Phi initial
LABEL
THETA
LAMBDA
COND
ATOM
EQ
QUOTE
NIL
FUNCTION
CAR
CDR
CONS
PHI
FN
List initial
Accu final

DEUXIEME PARTIE : LES ADDITIONS AU LISP "PUR".

Introduction.

Le LISP tel que nous l'avons défini dans la première partie permet, en principe, n'importe quel traitement de S-expressions. Toutefois, l'écriture de ce traitement peut être terriblement lourde et son exécution extrêmement lente. Nous allons indiquer trois raisons pour lesquelles le traitement en LISP "pur" est inefficient et la suite de cette deuxième partie contiendra l'exposé des solutions apportées en LISP "étendu".

La première raison est qu'une fonction ne peut être définie en LISP "pur" qu'au moment de son utilisation, ce qui provoque des notations aussi lourdes que celles du second exemple du chapitre I.6. Nous donnerons, au chapitre 1, le moyen de définir des fonctions utilisables, ensuite, par tous les programmes LISP suivants. Cette possibilité nouvelle permettra, non seulement d'augmenter la lisibilité d'un programme, mais aussi sa vitesse d'exécution.

La deuxième raison réside dans le nombre restreint de fonctions primitives. En ajoutant de " nouvelles fonctions primitives " à l'interpréteur, on pourra accélérer le traitement et gagner beaucoup de place en mémoire. En fait, ces fonctions pourront s'exprimer, sauf rares exceptions, en termes des cinq primitives de base, ce ne sont donc pas, à proprement parler, des "primitives", mais, parfois, leur expression en termes des fonctions primitives est tellement laborieuse que le gain obtenu est très considérable. (Voir à ce sujet, la fonction " REVERSE ".)

La troisième raison, c'est que le seul mécanisme que l'on ait à sa disposition, en LISP "pur", pour écrire un algorithme répétitif est la récursivité. Dans beaucoup de cas, l'emploi de la récursivité est extrêmement coûteux en temps et, surtout, en mémoire. Pour pouvoir écrire des boucles et des traitements semblables à ceux des langages séquentiels, on introduira, au chapitre 5, les formes spéciales de type PRØG, dont le rôle est, intuitivement, de déterminer une zone du programme dans laquelle on fait de la programmation séquentielle. L'utilisation systématique de cette troisième possibilité enlève beaucoup de son originalité au LISP : il n'est plus, alors, qu'un langage de manipulation de listes plus ou moins commode, mais il a entièrement perdu son caractère fonctionnel.

Nous verrons, en outre, dans ce chapitre, deux autres points plus particuliers, d'intérêt plus limité.

Le chapitre 3 introduit la notion de nombre, en LISP; l'utilité de cette extension n'est pas très considérable, mais cela permet de résoudre plus élégamment certains problèmes.

- Traitement de listes dont les éléments sont des nombres.
- Simplification de fonctions algébriques. (En effectuant les opérations qui ne portent que sur des nombres.)
- Compteurs de boucles, en liaison avec les formes de type PRØG.
- etc...

Enfin, dans le chapitre 4, nous introduirons un nouveau type de fonctions pour lesquelles le passage des arguments se fait par nom. Ceci est nécessaire lorsque les arguments ont une forme particulière et permet, notamment, de définir des fonctions dont le nombre d'arguments est quelconque. Cette notion permet également d'unifier la théorie des formes en LISP, conformément au § 2 du chapitre I.2.

Chapitre 1 : Les fonctions de type EXPR - Les constantes.

§ 1. "p-liste" d'un atome - pseudo-fonctions.

En LISP "pur", lorsque l'interpréteur rencontre une variable ou un identificateur de fonction, il doit parcourir la a-liste pour trouver la signification de ce symbole. Ce parcourt peut prendre un temps assez long.
Supposons que nous désirions donner à un symbole atomique une signification constante durant l'exécution des programmes suivants. Au lieu de placer ce symbole sur la a-liste (ou même, éventuellement, sur une "a'-liste" contenant les "constantes"), il est plus efficace de procéder de la façon suivante: dans la " représentation interne " de l'atome, on place un pointeur vers sa "valeur", s'il en a une, et un indicateur précisant de quel type de valeur, il s'agit. Le travail de l'interpréteur sera, alors, modifié: lors de la rencontre d'une variable ou d'une fonction atomique, il examinera si leur représentation interne contient un indicateur convenable, si oui, il accédera directement à la valeur et sinon il explorera la a-liste.

Nous donnerons la définition suivante:
on appelle p-liste (ou liste des propriétés) d'un atome, un assemblage de cellules permettant d'accéder

(1) à la suite des caractères de la représentation externe de l'atome,

(2) à une S-expression qui est la "valeur" ou encore la"définition", de l'atome,

(3) à un indicateur, donnant la nature de cette S-expression.

Les deux indicateurs qui nous intéresseront, ici, sont les symboles

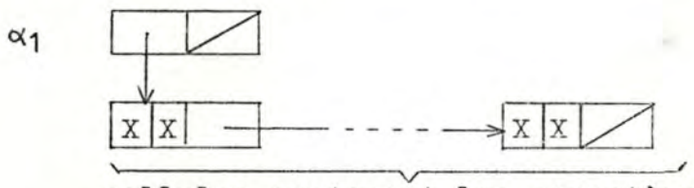
"EXPR" : indiquant que l'atome est un identificateur de fonction,

"APVAL" : indiquant que l'atome est une constante.

Si la S-expression (2) et l'indicateur (3) ne sont pas présents sur la p-liste du symbole, sa valeur doit être cherchée sur la a-liste.
Un symbole atomique est désigné, partout en mémoire, par l'adresse de la première cellule de sa p-liste. (cette adresse représente l'atome.)

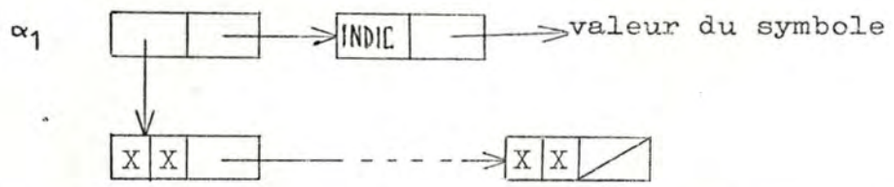
Format de la p-liste.

Dans notre système,
- lorsqu'on lit un atome pour la première fois, sa p-liste est créée avec le format suivant :



cellules contenant les caractères de la représentation externe de l'atome.

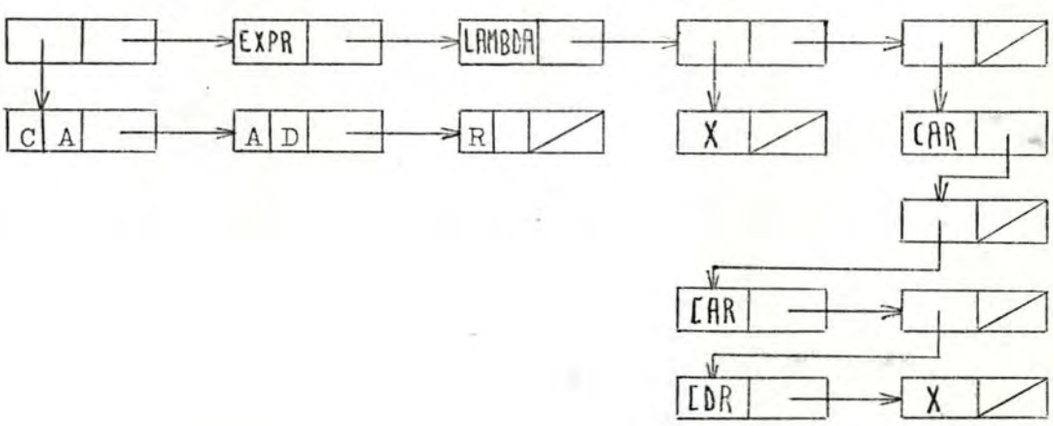
- si, par la suite, on a assigné au symbole une signification constante, le format de sa p-liste est le suivant :



Un symbole atomique est représenté partout en mémoire par l'adresse α_1 de la première cellule de sa p-liste; le symbole "INDIC" désigne l'un des indicateurs "APVAL" ou "EXPR".
Il est bon de remarquer que si α est l'adresse d'une S-expression, il faut maintenant effectuer le test "ALINK(ALINK(α))<0 ?" pour savoir si α est l'adresse d'un atome, au lieu de "ALINK(α)<0 ?" précédemment.

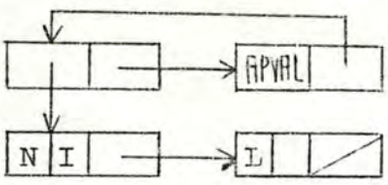
Exemples de p-listes.
---

(1)



L'indicateur "EXPR" dans la deuxième cellule de la p-liste indique que l'atome "CAADR" représente la fonction (LAMBDA(X)(CAR(CAR(CDR,X)))).

(2)



L'indicateur "APVAL" dans la deuxième cellule de la p-liste signifie que le symbole atomique "NIL" a pour valeur constante lui-même.

Pseudo-fonctions.

On appelle "pseudo-fonctions", en LISP, des fonctions qui modifient des structures en mémoire, par opposition aux autres fonctions qui créent de nouvelles structures, mais sans modifier celles déjà existantes. Le rôle de ces pseudo-fonctions étant de modifier des structures, la valeur renvoyée par l'une d'entre elles est généralement sans importance. D'où le nom de "pseudo-fonctions". Pour affecter une signification constante à un symbole, on doit modifier sa p-liste, ce qui nécessite l'utilisation d'une pseudo-fonction.

§ 2. La pseudo-fonction "DEFINE".

Cette pseudo-fonction a pour argument une liste de doublets:

$$((x_1^*,\varphi_1^*) \ldots (x_n^*,\varphi_n^*))$$

où les x_i^* sont des symboles atomiques

et les φ_i^* sont des fonctions, généralement de type λ .

L'effet de "DEFINE" est de placer l'indicateur "EXPR" sur la p-liste de chaque atome x_i^* ainsi que la représentation de la fonction φ_i^* correspondante. La valeur renvoyée par "DEFINE" est "NIL".

Par exemple, le programme LISP suivant:

(DEFINE((CAADR(LAMBDA(X)(CAR(CAR(CDR,X)))))))

(x_1^* ; φ_1^*)

a pour effet de créer la p-liste de l'exemple (1) de la page II.4.

§ 3. La pseudo-fonction "CSET".

Cette pseudo-fonction a deux arguments x* et α, x* doit être un atome et α est une S-expression quelconque. L'effet de "CSET" est de placer sur la p-liste de x* l'indicateur "APVAL" et la représentation de α . La valeur de "CSET" est x*.

Le programme

(CSET,NIL,NIL)

a pour effet de créer la p-liste de l'exemple (2) de la page II.5.

§4 . Modifications de l'interpréteur LISP.

Il faut, maintenant modifier l'interpréteur de façon à tenir compte de l'introduction de la notion de p-liste.

Les cas à modifier sont le cas A2 page I.35 et le cas E2 page I.38 ,

qui deviennent:

Cas A2 : φ^* est un symbole atomique.

PHI contient l'adresse de la p-liste du symbole φ^* et on a:

```
apply2 :    ACCU:=BLINK(PHI); [φ* est-il défini sur sa p-liste? ]
            si ALINK(ACCU)=EXPR, {PHI:=BLINK(ACCU);
                                  aller à Apply;
            ACCU:=ALIST; [sinon,recherche de la définition
                                                sur la a-liste.]
apply21 :   si ACCU=NIL, {imprimer "FØNCTIØN NØN DEFINIE";
                          ACCU:=PHI;   [erreur]
                          aller à findeevalquote;
            Q:=ALINK(ACCU);
            si ALINK(Q)=PHI, {PHI:=BLINK(Q);
                              aller à Apply;
            ACCU:=BLINK(ACCU);
            aller à apply21;
```

Cas E2 : ε^* est une variable.

EPSI contient l'adresse de la p-liste du symbole ε^* et on a l'algorithme suivant:

```
eval2 :     ACCU:=BLINK(EPSI); [ε* a sa valeur sur sa p-liste? ]
            si ALINK(ACCU)=APVAL, {ACCU:=BLINK(ACCU);
                                   aller à fin;
            ACCU:=ALIST; [sinon, recherche de la valeur sur
                                                la a-liste.]
eval21:     si ACCU=NIL, {imprimer "VARIABLE SANS VALEUR";
                          ACCU:=EPSI; [erreur.]
                          aller à findeevalquote;
            Q:=ALINK(ACCU);
            si ALINK(Q)=EPSI, {ACCU:=BLINK(Q);
                               aller à fin;
            ACCU:=BLINK(ACCU);
            aller à eval21;
```

§ 5. Un exemple : la fonction "RENVERSER".

Le programme LISP suivant conduit à la définition de trois fonctions de type EXPR: "LIST1" , "APPEND" , "RENVERSER".

```
(DEFINE(
(LIST1(LAMBDA(X)(CØNS,X(QUØTE,NIL))))
(APPEND(LAMBDA(X,Y)(CØND
                    ((ATØM,X)Y)
                    ((QUØTE,T)(CØNS(CAR,X)(APPEND(CDR,X)Y))))))

(RENVERSER(LAMBDA(X)(CØND
                     ((ATØM,X)X)
                     ((QUØTE,T)(APPEND(RENVERSER(CDR,X))
                                      (LIST1(CAR,X)))))))
))
```

Si on présente alors à l'interpréteur les 3 programmes

```
(LIST1,A)

(APPEND,(A,B,C,D)(E,F))

(RENVERSER(A,B,C,D,E,F))    ,
```

il renverra les résultats suivants:

```
(A)

(A B C D E F )

(F E D C B A ).
```

Chapitre 2 : Extension de la notion d'algorithme primitif:

Fonctions de type SUBR.

§ I. Introduction.

On peut, au moyen de la pseudo-fonction "DEFINE" définir autant de fonctions que l'on veut, utilisables par tout programme LISP ultérieur. Cependant, lorsqu'on fait appel très souvent à une fonction dans de nombreux types de traitements, il est avantageux de la définir par un sous-programme écrit en code machine, ou, dans notre cas, écrit en PL/I puis compilé, car une fonction de type EXPR est exécutée en mode interprétatif, ce qui nécessite une occupation mémoire plus grande et, surtout, un temps d'exécution plus long.

Dans la plupart des systèmes LISP, ces fonctions possèdent sur leur p-liste l'indicateur "SUBR" précisant qu'elles sont écrites en code machine et une adresse pointant vers le premier mot de mémoire du sous-programme définissant la fonction. Pour cette raison, on appelle ces fonctions, fonctions de type SUBR.

Les algorithmes primitifs du langage sont, eux-même, des fonctions de type SUBR et ces fonctions peuvent être considérées comme une extension de la notion d'algorithme primitif parce qu'on peut les utiliser sans les avoir, au préalable, définies. Toutefois, ces fonctions sont exprimables, la plupart du temps, en termes des 5 primitives de base.

Dans notre système, les fonctions de type SUBR ne sont pas reconnues par la présence de l'indicateur "SUBR" sur leur p-liste.
On procède de la façon suivante: les 50 premières cellules de la mémoire sont destinées à recevoir chacune la première cellule de la p-liste d'une fonction de type SUBR. Prenons un exemple, pour simplifier, et considérons la fonction "EQUAL" définie dans le corps de l'interpréteur par une séquence d'instructions précédée de l'étiquette " label(I9) : ". La première cellule de sa p-liste possède l'adresse I9 .
Nous modifions, en outre, très légèrement la première ligne de "apply" (page I.34) de la façon suivante:

Apply : si PHI≤50, aller à label (PHI);

Dés lors,l'utilisation de la fonction "EQUAL" dans un programme LISP donnera lieu à un branchement au début de "apply" avec PHI = 19,le test PHI≤50 étant positif,on se branchera à " label(19) " ,soit au début du sous-programme définissant "EQUAL".

§ 2. Définition de quelques fonctions de type SUBR .

Voici la définition de quelques fonctions de type SUBR qui existent dans notre système;nous n'écrirons pas pour toutes ces fonctions l'algorithme exact de représentation au sein de l'interpréteur,mais nous en donnerons,quand c'est possible,une définition en M-langage.(Celle-ci ne représentant pas l'algorithme effectif d'implémentation,mais définissant, sans ambiguïté,la sémantique de la fonction.)
Dans ce qui suit nous écrirons l'identificateur d'une fonction en lettres minuscules ou majuscules suivant qu'on utilise le M-langage ou le S-langage. Il est entendu,par exemple,que "null" et "NULL" désignent la même fonction : seules les notations diffèrent.

La fonction "APPEND"

admet deux arguments qui doivent être des listes. Elle a pour résultat une liste égale à la concaténation des deux arguments. Ceux-ci ne sont pas modifiés. On peut définir "APPEND" en termes du M-langage par l'équation :

$$\text{append}[x;y] = [\text{null}[x]\rightarrow y;T\Rightarrow\text{cons}[\text{car}[x];\text{append}[\text{cdr}[x];y]]] \quad .(*)$$

La fonction "APPLY"

est le sous-programme "apply" érigé en fonction utilisable par le programmeur LISP. Elle a trois arguments : une fonction,une liste de S-expressions et une liste de paires pointées;sa description est détaillée dans la première partie.

Considérons une fonction φ^* dans l'expression de laquelle interviennent n variables $x_1^*, \ldots, x_n^*$ libres et admettant m arguments.Dans ce cas,les deux programmes LISP suivants sont équivalents :

$$(\text{APPLY},\varphi^*(\alpha_1, \ldots, \alpha_m)((x_1^*.\alpha_{m+1}) \ldots (x_n^*.\alpha_{m+n})))$$

$$((\text{FUNARG},\varphi^*((x_1^*.\alpha_{m+1}) \ldots (x_n^*.\alpha_{m+n}))\ \alpha_1, \ldots, \alpha_m) \quad ,$$

où les α_i sont des S-expressions quelconques.

Les fonctions "CAAR" , "CADR" , "CDAR" , "CDDR" , "CADDR"

sont obtenues par composition des fonctions primitives "CAR" et "CDR" . La séquence de "A" et de "D" précise l'ordre de composition.

On a : $\text{caar}[x] = \text{car}[\text{car}[x]]$,

$\text{cadr}[x] = \text{car}[\text{cdr}[x]]$,

$\text{cdar}[x] = \text{cdr}[\text{car}[x]]$,etc ...

(*) La fonction "null" est définie ci-aprés.

La fonction "CØPY"

a pour argument une S-expression quelconque. Son résultat est une S-expression déduite de l'argument par "recopiage". On peut définir "CØPY" par :

copy[x] = [atom[x]→x;T→cons[copy[car[x]];copy[cdr[x]]]] .

Le prédicat "EQUAL"

teste si deux S-expressions quelconques sont égales, c'est à dire ont même représentation externe. Son résultat est fixé par l'égalité :

equal[x;y] = [atom[x]→[atom[y]→eq[x;y];T→F];
atom[y]→F;
equal[car[x];car[y]]→equal[cdr[x];cdr[y]];
T→F] .

La fonction "EVAL"

est la fonction correspondant au sous-programme "eval". Elle a normalement deux arguments : une forme et une liste de paires pointées.(*)

Si ε^* est une forme qui contient n variables libres $x_1^*, \ldots, x_n^*$, alors les deux programmes suivants sont équivalents :

(EVAL,ε^*(((x_1^*.α_1) ... (x_n^*.α_n)))

((LAMBDA(x_1^*, ... ,x_n^*)ε^*)α_1, ... ,α_n) .

Le prédicat "MEMBER"

admet deux arguments : une S-expression quelconque et une liste. Sa valeur est "T" si la S-expression est un élément de la liste, sinon sa valeur est "F". Sa définition en termes de M-langage est :

member[x;y] = [null[y]→F;
equal[car[y];x]→T;
T→member[x;cdr[y]]] .

La fonction "LIST"

admet un nombre indéfini d'arguments et a pour résultat la liste de ces arguments. On a :

list[x_1; ... ;x_n] = cons[x_1;cons[x_2; ... cons[x_n;NIL] ...]] .

(*) En fait, on peut utiliser "EVAL" avec le seul argument ε^*. Dans ce cas, on utilise la a-liste courante de l'interpréteur. De même, on peut utiliser "APPLY" avec pour seuls arguments une fonction φ^* et une liste de constantes (α_1, ... ,α_n) .

Le prédicat "NØT".

Normalement, ce prédicat a pour argument une valeur logique "T" ou "F". En fait, sa valeur est "T" si l'argument est "F" et dans tous les autres cas, sa valeur est "F". Sa définition en termes de M-langage est:

not[x] = [atom[x]→[eq[x;F]→T;T→F];T→F] .

Le prédicat " NULL "

a pour argument une S-expression quelconque, si c'est le symbole atomique "NIL", sa valeur est "T"; dans tous les autres cas, sa valeur est "F". Sa définition en termes de M-langage est:

null[x] = [atom[x]→[eq[x;NIL]→T;T→F];T→F] .

Les pseudo-fonctions "RPLACA" et "RPLACD"

ne sont pas réductibles à une combinaison des cinq fonctions primitives: elles modifient des structures préexistantes en mémoire. Elles ont deux arguments qui sont des S-expressions quelconques. On ne peut guère définir ces fonctions autrement qu'en faisant référence à la représentation interne des S-expressions, ce qui revient à dire qu'elles n'auraient probablement pas été introduites telles quelles si on avait fait choix d'une autre implémentation pour les listes.
Désignons, alors, par x le premier argument et par y le second et considérons la première cellule de la représentation de x en mémoire, cette cellule se compose de deux champs ALINK et BLINK;

"RPLACA" met l'adresse de y dans le champ ALINK,

"RPLACD" met l'adresse de y dans le champ BLINK.

La valeur de ces fonctions est l'argument x modifié.

Exemples.

- rplaca[(A,B,C,D);(E.F)] = ((E.F)B,C,D)
- rplacd[(A,B,C,D);(E.F)] = (A.(E.F))

La fonction "REVERSE"
---

a pour argument une liste, et pour résultat la liste obtenue en "renversant" l'ordre des éléments de la liste argument. Sa définition en termes de M-langage est:

reverse[x] = [null[x]→NIL;T→append[reverse[cdr[x]];list[car[x]]]] (1)

Cette façon de définir la fonction "REVERSE" par combinaison d'autres fonctions déjà définies n'est pas très naturelle. Si nous examinons la façon dont cette fonction est réalisée dans l'interpréteur, nous voyons que l'algorithme utilisé est bien plus simple que (1).

Les conditions initiales de l'algorithme sont les suivantes:

-LIST pointe vers la liste à "renverser",

-ACCU pointe vers le symbole "NIL".

Et on a l'algorithme:

```
reverse1 :   si LIST=NIL, aller à fin;
             EXFREE(ALINK(LIST),ACCU,ACCU);
             LIST:=BLINK(LIST); aller à reverse1;
```

Cet exemple montre qu'il est avantageux de coder directement certaines fonctions plutôt que de les définir à partir des primitives de base.Dans ce cas-ci, la raison pour laquelle la fonction "reverse" s'exprime aussi malaisément en fonction des primitives de base, c'est que le chaînage des cellules de listes n'est réalisé que dans un sens: de la gauche vers la droite et que pour construire la première cellule du résultat, il faut connaître la dernière cellule de la liste à renverser; or, une fois qu'on connaît cette cellule, on ne peut plus accéder aux cellules précédentes sauf si on les a mémorisées dans un stack, ce qui est réalisé dans (1) par l'appel récursif de "reverse". On est obligé d'agir ainsi parce qu'on ne possède pas d'"instruction d'affectation" permettant de stocker dans une variable des résultats intermédiaires ni d'"instruction de branchement" permettant d'effectuer une boucle. Malheureusement l'introduction de ces mécanismes détruit le caractère fonctionnel du LISP. (Voir chapitre 5).

Chapitre 3 : Les nombres.

Nous ne discuterons pas longuement de l'utilité d'introduire la notion de nombre en LISP.Il est certain que ce langage n'est pas bien adapté à la réalisation d'algorithmes numériques.Toutefois,cette notion peut rendre plus élégante la résolution de certains problèmes.Nous l'utiliserons dans le second exemple de la cinquième partie.

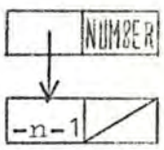
§ 1. Représentation externe d'un nombre-Représentation interne .

Dans notre système,un nombre est une suite de 1 à 5 chiffres représentant un entier compris entre 0 et 32.766. On décide de noter les nombres par une lettre "n" indicée ou non.

Les nombres sont des symboles atomiques,bien que d'un type particulier.Cela veut dire que si n est un nombre,

atom[n] = T .

Pour répondre à cette exigence,la p-liste d'un nombre n a reçu le format ci-dessous :



où la case NUMBER contient l'adresse du symbole atomique "NUMBER" et la case -n-1 contient une configuration de bits qui est la représentation du nombre entier -n-1 dans le code de la machine .

En définissant les nombres de cette manière,on voit que le test d'atomicité :

si α est l'adresse d'un atome, ALINK(ALINK(α))<0 ,

est vérifié aussi pour les nombres.

Rôle de l'identificateur "NUMBER".

Cet indicateur distingue un nombre d'un atome qui n'est pas un nombre.A ce sujet,on peut faire l'importante remarque suivante :

Si α est l'adresse d'une p-liste,le champ ALINK(ALINK(α)) doit être interprété différemment suivant la valeur du champ BLINK(α) :

- si BLINK(α)=NUMBER , ALINK(ALINK(α)) contient la représentation d'un nombre n sous la forme -n-1 ,dans le code de la machine,

- si BLINK(α)$\neq$NUMBER , ALINK(ALINK(α)) contient la représentation des deux premiers caractères d'un atome dans le code de la machine .

§ 2. Valeur d'un nombre - Modification de l'interpréteur.

La représentation d'un nombre est la même en M-langage et en S-langage.Cela ne provoque pas d'ambiguïté car un nombre considéré comme un identificateur à toujours pour valeur lui-même,c'est donc une constante semblable à celles définies par la présence de l'identificateur "APVAL" sur leur p-liste.
Pour tenir compte de cette nouvelle notion,on modifie,encore une fois,le texte de l'interpréteur.C'est le cas E2 qui doit être transformé : on introduit entre la ligne 1 et la ligne 2 de la page II.6 ,le texte suivant :

si ACCU=NUMBER, { ACCU:=EPSI;
                { aller à fin;

On peut constater,maintenant,que l'évaluation par "eval" des deux S-expressions

(QUØTE,123) et 123

donnera chaque fois le résultat "123" .

§ 3. Les fonctions de type SUBR relatives aux nombres.

Les prédicats "NUMBERP" et "EQUAL" .

Le prédicat "NUMBERP" permet de distinger, parmi les symboles atomiques,ceux qui sont des nombres.

Si x* est un atome,

(NUMBERP,x*) est T si x* est un nombre

et F si x* n'est pas un nombre,

si on applique "NUMBERP" à un argument qui n'est pas un atome ,sa valeur est indéfinie.
On peut encore définir "NUMBERP" par l'égalité du M-langage :

numberp[x] = eq[cdr[x];NUMBER] .

Le prédicat "EQUAL" doit être modifié de manière à tenir compte de la notion de nombre.Dans la définition antérieure que nous avons donnée de "EQUAL" ,on se basait sur le prédicat "eq" pour établir l'égalité de deux atomes,et "eq" se contente d'examiner si les adresses de ces deux atomes sont identiques.Or,on verra,d'après les opérations définies sur les nombres qu'à la suite de certaines manipulations,il se peut qu'un même nombre soit représenté plusieurs fois en mémoire. Il est,évidemment,inadmissible de considérer ces différentes représentations comme des êtres distincts.

C'est pourquoi,nous redéfinissons la fonction "equal" par l'équation suivante du M-langage :

```
equal[x;y] =
[atom[x]→[atom[y]→[eq[x;y]→T;
                   numberp[x]→[numberp[y]→[lessp[x;y]→F;
                                           T→zerop[difference[x;y]]];
                               T→F];
                   T→F];
          T→F];
 atom[y]→F;
equal[car[x];car[y]]→equal[cdr[x];cdr[y]];
 T→F]     ,
```

On trouvera ci-après les définitions des fonctions "lessp" , "zerop" et "difference".

Pseudo-fonctions arithmétiques : "ADD1" et "SUB1" .

Si n est un nombre dont α est l'adresse de la p-liste,

"ADD1" décrémente ALINK(ALINK(α)) de 1 .

"SUB1" incrémente ALINK(ALINK(α)) de 1 .

Dès ce moment,le nombre n n'a plus de représentation en mémoire.

Fonctions arithmétiques :

"PLUS" , "TIMES" ,"QUØTIENT" , "DIFFERENCE" , "DUP" , "EXPT".

Si on a p nombres $n_1, \ldots, n_p$,

(PLUS,n_1, ... ,n_p) est le nombre $n_1 + \ldots + n_p$,

(TIMES,n_1, ... ,n_p) est le nombre $n_1 * \ldots * n_p$,

à condition que ces nombres soient compris entre 0 et 32.766 .

Si n_1 et n_2 sont deux nombres,

(QUØTIENT,n_1,n_2) est la partie entière de n_1/n_2 , à condition que $n_2 \neq 0$,

(DIFFERENCE,n_1,n_2) est le nombre $n_1 - n_2$ si $n_1 \geqslant n_2$ et est indéfini sinon .

(EXPT,n_1,n_2) est le nombre $n_1^{n_2}$, à condition que ce nombre soit inférieur à 32.766.

Si n est un nombre,

(DUP,n) est une "copie" de ce nombre.

Prédicats arithmétiques : "ZERØP" , "LESSP" , "GREATERP" .

Si n_1 et n_2 sont des nombres,

(ZERØP,n_1) a pour valeur "T" si $n_1 = 0$ et "F" sinon ,

(LESSP,n_1,n_2) a pour valeur "T" si $n_1 < n_2$ et "F" sinon ,

(GREATERP,n_1,n_2) a pour valeur "T" si $n_1 > n_2$ et "F" sinon.

Les fonctions "LENGTH" et "MINUS" .

La fonction "LENGTH" a pour argument une liste et pour valeur le nombre d'éléments de cette liste.On peut la définir par l'équation du M-langage :

```
length[x] = [null[x]→dup[0];T→add1[length[cdr[x]]]] .
```

La fonction "MINUS" a pour argument soit un nombre,soit une liste du type (MINUS,n).On définit cette fonction par l'équation :

```
minus[x] = [atom[x]→[zerop[x]→0;T→list[MINUS;x]];T→cadr[x]].
```

Chapitre 4. : Les fonctions de type FSUBR et FEXPR.

§ 1. Introduction : Le symbole "CØND" considéré comme un identificateur de fonction.

Jusqu'ici, nous avons considéré, principalement, deux types de formes.

- Les applications de fonction, comme, par exemple, (CØNS,X,Y) ,

- Des"formes spéciales", comme, par exemple, (CØND((ATØM,X)X)((QUØTE,T)(CDR,X))).

Or, ce second type de forme peut, lui aussi, être considéré comme l'application d'une fonction à des arguments. Ainsi l'exemple donné plus haut, peut être regardé comme l'application de la fonction "CØND" aux deux arguments ((ATØM,X)X) et ((QUØTE,T)(CDR,X)). Ce qui distingue les deux types de formes, c'est la façon de transmettre les arguments à la fonction.

- Dans le cas de (CØNS,X,Y), les arguments sont tous évalués dans un premier temps, par appel du sous-programme "eval"; ensuite, les valeurs de ces arguments sont transmises au sous- programme correspondant à "CØNS" qui les manipule de manière à fournir le résultat.

- Dans le cas de (CØND((ATØM,X)X)((QUØTE,T)(CDR,X))), les arguments sont transmis au sous-programme correspondant à "CØND" sans être évalués: c'est à l'intérieur de ce sous-programme qu'on évaluera tous ou une partie de ces arguments.
En résumé, on peut dire que:

-"CØNS" est une fonction dont les arguments sont passés par valeur: elle est du type SUBR.

-"CØND" est une fonction dont les arguments sont passés par nom. Elle est du type FSUBR.

Nous verrons dans le paragraphe suivant les fonctions de type FSUBR qui existent dans notre système, ce qui montrera l'intérêt de cette notion. Remarquons, dès maintenant, qu'elle unifie la théorie des formes puisque toute forme pourra être considérée soit comme une variable, soit comme l'application d'une fonction.

§ 2. Les fonctions de type FSUBR.-

On appelle fonctions de type FSUBR, des fonctions qui sont incorporées à l'interpréteur et qui se caractérisent par le fait que leurs arguments sont passés par nom au sous-programme qui leur correspond. C'est à l'intérieur de ce sous-programme qu'on décide de l'évaluation des arguments.

Dans la plupart des systèmes LISP, ces fonctions sont reconnues par la présence de l'indicateur "FSUBR" dans leur p-liste. Dans notre système, on procède autrement:
chacune des cellules de mémoire d'adresses 5I à 70 est réservée à la première cellule d'une p-liste de fonction de type FSUBR. Cela donne un moyen simple de reconnaître ces fonctions.

Les fonctions "QUØTE", "CØND", "FUNCTIØN",

sont du type FSUBR; leur signification a été vue plus haut, bien que d'un point de vue différent.

Les fonctions logiques "AND" et "ØR",

sont du type FSUBR et admettent un nombre indéfini d'arguments qui sont, théoriquement, des formes prédicatives $\varepsilon_1^*, \ldots, \varepsilon_n^*$.

Le prédicat "AND" évalue successivement les arguments ε_i^* à partir du premier, jusqu'à en trouver un dont la valeur est le symbole "F". Si on trouve un tel argument, le résultat final est le symbole "F" et les arguments restants ne sont pas évalués, sinon la valeur de "AND" est le symbole "T".

Le prédicat "ØR" évalue successivement les arguments à partir du premier jusqu'à en trouver un dont la valeur est différente de "F". Dans ce cas, le résultat final est le symbole "T" et sinon, "ØR" renvoie la valeur "F" après évaluation de tous les arguments.

La pseudo-fonction "CSETQ"

a pour arguments, un symbole atomique x* et une forme ε^* et les deux formes

(CSET(QUØTE,x*)ε^*) et (CSETQ,x*,ε^*)

sont équivalentes.

§ 3. Les fonctions de type FEXPR.

On introduit ces fonctions pour pallier deux insuffisances des fonctions de type EXPR.

- Une fonction de type EXPR a toujours un nombre d'arguments fixé par sa définition.

- Les arguments d'une fonction de type EXPR sont tous évalués avant d'être manipulés par la fonction.

Une fonction de type FEXPR se caractérise par la présence sur sa p-liste l'indicateur "FEXPR" associé à une fonction-λ qui définit le traitement effectué par la fonction de type FEXPR. L'interprétation de cette fonction-λ est parfaitement différente de celle des fonctions-λ de définition des fonctions de type EXPR. En effet, elle a toujours exactement deux arguments:

- le premier est la liste des arguments non évalués de la fonction de type FEXPR associée,

- le second est une liste de paires pointées: généralement, la a-liste qui servira à évaluer les éléments de la liste, premier argument.

Cette différence se marque de la façon suivante dans le fonctionnement de l'interpréteur:

Considérons une forme du type $(\varphi^*, \varepsilon_1^*, \ldots, \varepsilon_n^*)$ où φ^* est une fonction de type EXPR ou FEXPR et soit ψ^* la fonction-λ associée.

Si φ^* est de type EXPR,

"eval" évalue les n arguments $\varepsilon_1^*, \ldots, \varepsilon_n^*$,
constitue la liste $(\alpha_1, \ldots, \alpha_n)$ de leur valeurs,
passe la main à "apply" avec les 3 arguments

- φ^*
- $(\alpha_1, \ldots, \alpha_n)$
- a .

Si φ^* est de type FEXPR,

"eval" constitue la liste à 2 éléments $((\varepsilon_1^*, \ldots, \varepsilon_n^*)a)$,
recherche la fonction ψ^* associée,
passe la main a "apply" avec les 3 arguments

- ψ^*
- $((\varepsilon_1^*, \ldots, \varepsilon_n^*)a)$
- a .

Exemple.

Si l'on a une fonction de type FEXPR, et qu'on la transforme simplement en mettant l'indicateur "EXPR" sur sa p-liste, la fonction obtenue aura une signification tout à fait différente de la première et produira généralement un résultat imprévisible.

Nous donnons toutefois, un exemple construit spécialement pour obtenir un résultat valide dans les deux cas et montrer qu'ils sont distincts.
Supposons que nous présentions à l'interpréteur la séquence suivante de programmes LISP:

```
(DEFINE(                                                              (1)
(MAPLIST(LAMBDA(X,FN)(CØND
                      ((NULL,X)X)
                      ((QUØTE,T)(CØNS(FN,X)
                                     (MAPLIST(CDR,X)
                                              (FUNCTIØN,FN)))))))

(LISTE(LAMBDA(X,Y)(MAPLIST,X(FUNCTIØN
                             (LAMBDA(X)(EVAL(CAR,X)Y))))))
))

(LISTE(QUØTE,LAMBDA)((LAMBDA,NIL(QUØTE,NIL))(QUØTE,NIL)))             (2)

((LAMBDA(X,Y)(RPLACA(CDR,X)Y))LISTE,FEXPR)                            (3)

(LISTE(QUØTE,LAMBDA)((LAMBDA,NIL(QUØTE,NIL))(QUØTE,NIL)))  .          (4)
```

Le programme (1) provoque la création de la fonction "LISTE" du type EXPR.

Le programme (2) provoque l'évaluation par "LISTE" des éléments de la liste (QUØTE,LAMBDA) avec la liste de paires pointées

```
((LAMBDA.(NIL(QUØTE,NIL)))(QUØTE.(NIL)))  .
```

Le résultat est la liste des valeurs trouvées, c'est à dire:

```
((NIL)(NIL(QUØTE,NIL)))                                               (RI)
```

Le programme (3) a pour effet de transformer "LISTE" en fonction de type FEXPR.

Le programme (4) provoque l'évaluation des deux formes

```
(QUØTE,LAMBDA) et ((LAMBDA,NIL(QUØTE,NIL))(QUØTE,NIL))
```

avec une a-liste vide. Le résultat est la liste des valeurs trouvées, c'est à dire:

```
(LAMBDA,NIL)                                                          (R2)
```

Les résultats (RI) et (R2) sont bien différents et c'est ce que nous voulions montrer.

Chapitre 5. : Les formes spéciales de type PRØG.

§1. Introduction .

Nous allons introduire la possibilité d'utiliser en LISP des mécanismes de programmation séquentielle:

- séquences d'instructions
- affectations
- branchements .

L'avantage de cette introduction, c'est qu'un grand nombre de traitements pourront être réalisés beaucoup plus aisément. Mais le prix de cette amélioration est que le LISP ainsi étendu, n'est plus un langage fonctionnel et il a perdu sa plus grande qualité qui était sa simplicité. En particulier, la description entièrement formelle de sa sémantique est certainement beaucoup moins simple à écrire.

En guise d'introduction, nous écrirons une procédure Algol qui calcule le plus grand commun diviseur de deux nombres entiers, suivie d'une fonction LISP équivalente. Nous donnerons ensuite une description rigoureuse des formes de type PRØG.

Procédure Algol .

```
integer procedure pgcd(n,m);
                  integer n,m;
     begin integer r,q;
     loop :       q:=n'/.m; r:=n-q*m;
                  if r=0 then goto exit;
                  n:=m; m:=r; goto loop;
     exit :       pgcd:=m;
end end
```

Fonction LISP .

```
(PGCD(LAMBDA(N,M)
     (PRØG(R,Q)
      LØØP       (SETQ,Q(QUØTIENT,N,M))
                 (SETQ,R(DIFFERENCE,N(TIMES,Q,M)))
                 (CØND((ZERØP,R)(RETURN,M)))
                 (SETQ,N,M)
                 (SETQ,M,R)
                 (GØQ,LØØP)
)))
```

La comparaison de la fonction LISP avec le programme Algol permet de réaliser intuitivement la signification des nouvelles fonctions LISP introduites.

- "SETQ" : affectation d'une valeur à une variable.
- "GØQ" : branchement .
- "RETURN" : sortie de la forme de type PRØG .

§ 2. Syntaxe des formes de type PRØG .

Une forme de type PRØG est une liste dont

le premier élément est le symbole "PRØG" ,

le deuxième élément est une liste de variables appelées "variables programme",

les éléments suivants, en nombre quelconque, sont

soit des étiquettes , c'est à dire des atomes,

soit des instructions, c'est à dire des formes non atomiques.

§ 3. Nature des instructions.

Il y a des restrictions sur la nature des formes admissibles comme instructions. Les formes suivantes sont des instructions valides.

(1) Toute forme non atomique, valide en LISP avant l'introduction des formes de type PRØG .

(2) Des formes du type

(GØ, ε^*) où ε^* est une forme qui n'est pas du type (2),

ou du type

(GØQ, x^*) où x^* est une étiquette,

ou du type

(RETURN, ε^*) où ε^* est une forme qui n'est pas du type (2) .

(3) Des formes du type

(SET, ε_1^*, ε_2^*) où ε_1^* et ε_2^* sont des formes qui ne sont pas du type (2) ,

ou du type

(SETQ, x^*, ε^*) où x^* est une variable et ε^* est une forme qui n'est pas du type (2),

ou du type

(CØND(ε_1^*, ω_1^*) ... (ε_n^*, ω_n^*)) où les ε_i^* sont des formes qui ne sont pas du type (2) et les ω_i^* sont des formes qui sont de n'importe lequel des types (1) à (5).

(4) Des formes du type PRØG.

(5) Des formes construites à partir des formes des types (1) , (3) , (4) , (5) et des règles habituelles de construction des formes.

La raison pour laquelle on introduit ces restrictions, c'est qu'on a voulu éviter, au maximum, les situationsdifficiles à implémenter. Nous verrons, dans la suite, une autre restriction introduite pour la même raison.

§ 4. "Entrée" dans une forme du type PRØG.

Lorsque le sous-programme "eval" commence l'évaluation d'une forme du type PRØG, il effectue d'abord une série d'opérations préliminaires.

- Mémorisation de la valeur actuelle de la a-liste et de la valeur actuelle d'une autre liste de paires pointées appelée "GØ-liste".

- Modification de la a-liste: soit $(x_1^*, \ldots, x_n^*)$ la liste des variables programme, on ajoute les n paires pointées $(x_n^*.NIL), \ldots, (x_1^*.NIL)$ au début de la a-liste.

- Modification de la GØ-liste: on parcourt toute la liste constituée par la forme du type PRØG, chaque fois qu'on rencontre une étiquette, on rajoute au début de la GØ-liste une paire pointée dont le premier élément est cette étiquette et le second est la liste des éléments suivants de la forme de type PRØG.

§ 5. Evaluation d'une forme du type PRØG.

Lorsque cette phase préliminaire est terminée, "eval" évalue les instructions en séquence jusqu'à ce qu'un des trois événements suivants se produise.

- Rencontre d'une instruction (GØ, ε*) ou (GØQ, x*). Dans ce cas, on cherche sur la GØ-liste la première paire pointée ayant pour premier élément x* ou eval [ε*;a] . Le second élément de cette paire pointée est une liste d'instructions: on continue l'évaluation séquentielle à partir de la première instruction de cette liste. (*) Si une telle paire pointée n'est pas trouvée, il y a impression d'un libellé d'erreur et l'exécution se termine.

(*) Si l'on a des formes de type PRØG, imbriquées, (comme des procédures en Algol) On ne peut sortir d'une forme de type PRØG par un branchement vers une étiquette d'une forme de type PRØG extérieure. Ici encore, c'est pour simplifier l'implémentation qu'on a pris cette option.

- Rencontre d'une instruction du type (RETURN,ε^*). On évalue, alors, la forme ε^* par appel de "eval"; ensuite, on restaure la a-liste et la GØ-liste telles qu'elles étaient avant l'évaluation de la forme de type PRØG, dont la valeur est, finalement, la valeur de ε^*.

- On a fini d'évaluer la dernière instruction de la forme de type PRØG et ce n'est pas un branchement. Dans ce cas, on effectue une action équivalente à la rencontre de l'instruction

(RETURN(QUØTE,NIL)) .

Pour être complet, il reste à définir les <u>instructions d'affectation.</u>

- L'instruction (SET,ε_1^*,ε_2^*) provoque l'évaluation des formes ε_1^* et ε_2^*.
La valeur de ε_1^* doit être un symbole atomique.
On cherche ensuite la première paire pointée de la a-liste dont le premier élément est ce symbole atomique. Si cette paire est trouvée, on remplace son second élément par la valeur de ε_2^*, sinon, on imprime un libellé d'erreur et l'exécution s'arrête. La valeur de la forme (SET,ε_1^*,ε_2^*) est la valeur de ε_2^*.
On voit que "SET" est une pseudo-fonction.

- L'instruction (SETQ,x*,ε^*) a une action équivalente à l'instruction

(SET(QUØTE,x*)ε^*).

Chapitre 6 : Deux exemples : addition de deux listes de chiffres.

Dans les pages qui suivent,nous donnons deux exemples de programmes LISP effectuant le même type de traitement,à savoir l'addition de deux listes représentant des nombres entiers.

Le premier exemple ne se sert pas des fonctions de manipulation des nombres et ne comporte pas de forme de type PRØG ; d'ailleurs,les chiffres sont représentés par les atomes "ZERØ" , "UN" , ... , "NEUF". Cet exemple ultra récursif,nécessite beaucoup de temps et de mémoire . (Utilisation du "Garbage collector" .)

Le second exemple,utilisant les fonctions arithmétiques et construit sur la base d'une forme de type PRØG,est de loin plus économique en temps et en mémoire : les listes traitées ont plus d'éléments et le Garbage collector n'est pas utilisé.

Ces exemples sont tirés de la référence [7]. Certaines modifications mineures ont été effectuées.

```
   (DEFINE(
   (SUM(LAMBDA(X Y)(REVERSE(SUMREV(REVERSE X)(REVERSE Y)ZERO))))
   (SUMREV(LAMBDA(U V P)(COND((EQ U NIL)(COND((EQ V NIL)
   (COND((EQ P ZERO)NIL)(T(CONS UN NIL)))))(T(COND((EQ P ZERO)V)
   (T(SUMREV(CONS ZERO NIL)V UN))))))
   ((EQ V NIL)(COND((EQ P ZERO)U)(T(SUMREV U(CONS ZERO NIL)UN))))
   (T(CONS(SUMD(CAR U)(CAR V)P)(SUMREV(CDR U)(CDR V)
   (REPORT(CAR U)(CAR V)P)))))))
   (SUC(LAMBDA(N)(COND((EQ ZERO N)UN)((EQ UN N)DEUX)((EQ DEUX N)TROIS)
   ((EQ TROIS N)QUATRE)((EQ QUATRE N)CINQ)((EQ CINQ N)SIX)
   ((EQ SIX N)SEPT)((EQ SEPT N)HUIT)((EQ HUIT N)NEUF)((EQ NEUF N
   )ZERO))))
   (PRE(LAMBDA(  N)(COND((EQ ZERO N)NEUF)((EQ UN N)ZERO)
   ((EQ DEUX N)UN)((EQ TROIS N)DEUX)((EQ QUATRE N)TROIS)((EQ CINQ N)
   QUATRE)((EQ SIX N)CINQ)((EQ SEPT N)SIX)((EQ HUIT N)SEPT)((EQ NEUF N)
   HUIT))))
   (DIR(LAMBDA(A,B)(COND((EQ ZERO A)B)((EQ ZERO(SUC B))(PRE A))
   (T(DIR(PRE A)(SUC B))))))
   (IND(LAMBDA(A,B)(COND((EQ ZERO A)ZERO)((EQ ZERO(SUC B))UN)
   (T(IND(PRE A)(SUC B))))))
   (SUMD(LAMBDA(A,B,C)(DIR(DIR A B)C)))
   (REPORT(LAMBDA(A,B,C)(COND,
   ((EQ(IND A B)UN)UN)(T(IND(DIR A B)C)))))
   ))
   DEBUT DE EVALQUOTE
   FIN DE EVALQUOTE
NIL

   (SUM(QUATRE TROIS SEPT)(CINQ NEUF UN))
   DEBUT DE EVALQUOTE
   FIN DE EVALQUOTE
( UN ZERO DEUX HUIT )

   (SUM(UN DEUX TROIS QUATRE CINQ SIX SEPT HUIT NEUF )
       (DEUX QUATRE SIX HUIT NEUF SEPT CINQ TROIS UN)        )
   DEBUT DE EVALQUOTE
   GARBAGE COLLECTOR
   FIN DE EVALQUOTE
( TROIS  SEPT ZERO TROIS  CINQ QUATRE TROIS  DEUX ZERO )
   EJECT
```

```
(DEFINE(
(APPEND(LAMBDA(U,V)(COND,
                    ((NULL U)V),
                    (T(CONS(CAR U)(APPEND(CDR U)V))))))
(SUM(LAMBDA( X Y)(SUMOR(REVERSE X)(REVERSE Y)0)))
(SUMOR    (LAMBDA(X Y R)(PROG(U W)
   D     (COND((NULL X)(GO (QUOTE A)))
               ((NULL Y)(GO (QUOTE B)))
               (T NIL))
         (SET(QUOTE U)(PLUS(CAR X)(CAR Y)R))
         (SET(QUOTE X)(CDR X))
         (SET(QUOTE Y)(CDR Y))
         (SET (QUOTE R)(DUP 0))
         (COND((LESSP U 10)(GO(QUOTE C)))
                        (T NIL))
         (SET(QUOTE U)(DIFFERENCE U 10))
         (SET(QUOTE R)(DUP 1))
   C     (SET(QUOTE W)(CONS U W))
         (GO(QUOTE D))
   A     (COND((ZEROP R)(RETURN (APPEND(REVERSE Y)W)))
               ((NULL Y)(RETURN(CONS (DUP 1)W)))
               (T NIL))
         (SET(QUOTE X)(LIST(DUP 1)))
         (SET(QUOTE R)(DUP 0))
         (GO (QUOTE D))
   B     (COND((ZEROP R)(RETURN(APPEND(REVERSE X)W)))(T NIL))
         (SET(QUOTE Y)(LIST (DUP 1)))
         (SET(QUOTE R)(DUP 0))
         (GO (QUOTE D)))))
))
DEBUT DE EVAL QUOTE
FIN DE EVALQUOTE
NIL

(SUM
(9 9 9 8 8 8 7 7 7 6 6 6 5 5 5 4 4 4 3 3 3 2 2 2 1 1 1 )
(1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 )
)
DEBUT DE EVAL QUOTE
FIN DE EVALQUOTE
( 1 1 2 3 3 4 5 5 6 6 7 9 0 0 1 2 2 3 3 4 5 6 6 7 8 9 0 0 )
EJECT
```

TROISIEME PARTIE : LES ROUTINES D'ENTREE / SORTIE ET LE SUPERVISEUR.

Chapitre 1 : La notion de "table des symboles" - Réalisation dans notre système.

Lorsqu'un compilateur effectue la traduction d'un programme écrit en langage évolué, une de ses premières tâches est de construire une table dont chaque entrée correspond à un identificateur existant dans le texte source. Généralement, on crée pour cet identificateur une entrée dans la table la première fois qu'on le rencontre et, au fur et à mesure de l'analyse du texte source, on rajoute dans la table des renseignements relatifs à sa nature. En particulier, il est important de tenir compte du fait que toutes les occurences d'un identificateur dans le texte source sont des références à un même être, il ne s'agit donc pas de créer une entrée dans la table chaque fois qu'on rencontre l'identificateur, mais il faut posséder un algorithme permettant de déterminer s'il est déjà représenté dans la table ou pas.
On peut, alors, remplacer toutes les occurences de l'identificateur dans le texte source par un pointeur vers l'entrée qui lui correspond dans la table.

En LISP, on effectue aussi une traduction, assez élémentaire, il est vrai, du texte source, puisqu'on représente les S-expressions, en mémoire, par des arbres binaires. Lors de cette traduction, on crée pour chaque atome une entrée dans la table des symboles qui n'est rien d'autre que sa p-liste. Celle-ci est unique pour un atome donné et, de plus, l'atome est représenté dans l'arbre binaire correspondant à la traduction du texte source par un pointeur vers le début de sa p-liste. Il est donc nécessaire d'avoir un algorithme permettant de retrouver la p-liste d'un atome connaissant sa représentation externe. La technique que nous utilisons est assez rudimentaire mais d'une grande simplicité et d'une grande souplesse, car la table des symboles n'est pas localisée dans une zone fixée de la mémoire.

On divise l'ensemble des symboles atomiques possible en sept familles:

- les nombres,
- les atomes de longueur $\leq 2, 3, 4, 5, 6, \geq 7$ caractères.

A chacune de ces familles, on associe, à tout moment, une liste d'accès permettant d'atteindre les p-listes de tous les atomes de la famille rencontrés précédemment. L'adresse de la première cellule de chaque liste d'accès est connue et fixe.

Dès lors, chaque fois qu'on lit un atome dans le texte source,

on calcule sa longueur,

on détermine la liste d'accès qui lui correspond,

on parcourt cette liste,

on accède aux p-listes des atomes déjà représentés,

on compare, pour chaque p-liste, la séquence de caractères qu'elle contient avec la séquence de caractères du symbole lu,

s'il y a identité, on a déterminé la p-liste de l'atome;

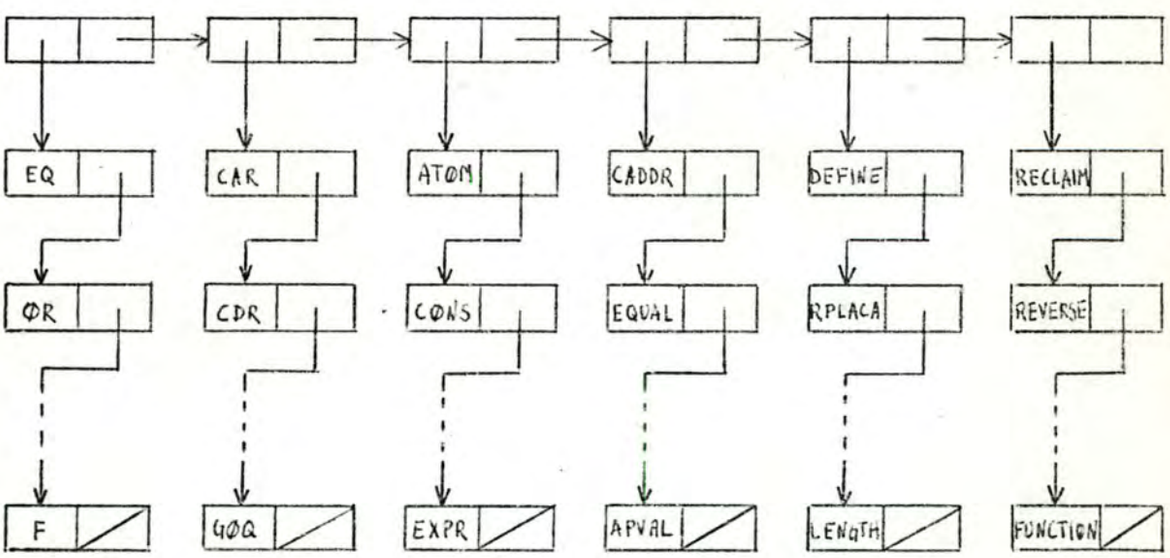
si on ne trouve aucune p-liste pour laquelle il y a identité,

on crée une p-liste pour le symbole,

on rajoute un élément à la liste d'accès permettant d'atteindre cette p-liste.

A l'initialisation de l'interpréteur, un certain nombre de p-listes sont créées d'office dans la zone de mémoire réservée et les listes d'accès sont agencées suivant le schéma de la figure III.1.

figure III.1



Chapitre 2 : Le programme de lecture.

Ce programme a pour fonction de lire une S-expression sur le support d'entrée et de la représenter en mémoire sous forme d'arbre binaire. Nous décrivons, ci-après, les différents sous-programmes qui le constituent.

§ 1. Le sous-programme "READ".

Le texte source est décomposé physiquement en blocs de 80 caractères, sur le support d'entrée. Ce sous-programme lit l'enregistrement "suivant" sur le support d'entrée, le stocke dans une zone de mémoire nommée BUF1 et l'imprime sur le support de sortie.

§ 2. Le sous-programme "LIRSYMB"

a pour rôle de lire le "symbole suivant" du texte source considéré comme un flot continu de caractères. Il utilise la fonction "SUBSTR" de trois arguments X , Y , Z qui sélectionne dans la chaîne de caractères X , la sous-chaîne de longueur Z qui commence au Y-ième caractère.
Le symbole lu est placé dans la position de mémoire S .

```
sous-programme LIRSYMB;
               si K>80, {READ;
                        {K:=1;
               S:=SUBSTR(BUF1,K,1);
               fin;
```

§ 3. Le sous-programme "LIRATØM"

examine si un symbole atomique dont la représentation externe commence au K-ième caractère de BUF1 est déjà représenté en mémoire, sinon, il crée la p-liste de ce symbole. Dans tous les cas, il met dans le pointeur SØ l'adresse de la p-liste du symbole atomique.
Ce sous-programme utilise un tableau à I6 éléments de 2 caractères, nommé BUF, destiné à recevoir les caractères du symbole examiné. Pour cette raison, dans notre système, un symbole atomique a une longueur limitée à 32 caractères.
Le tableau à 6 éléments LISTI , contient les adresses des listes d'accès aux p-listes des atomes.

```
sous-programme LIRATØM;
               L:=K;
a :            K:=K+1; [calcul longueur atome.]
               S:=SUBSTR(BUF1,K,1);
               si "A"≤S≤"Z" ou "0"≤S≤"9", aller à a;
               M:=K-L; K:=K-1; N:=(M+1)*/.2;
               si M≤2, RS:=LISTI(1); [choisir liste d'accès.]
               sinon {si M≥7, RS:=LISTI(6);
                      {sinon RS:=LISTI(M-1);
               I:=1;[ mettre dans BUF, les caractères du symbole.]
               effectuer N fois {BUF(I):=SUBSTR(BUF1,L,2);
                                {I:=I+1; L:=L+2;
               si M≠2*N, SUBSTR(BUF(N),2,1):=" ";
c :            SØ,R:=ALINK(RS); [recherche du symbole dans
                                    la liste d'accès.]
               R:=ALINK(R);I:=1;
ca :           L:=BLINK(R);
               si INFØ(R)=BUF(I), { si I=N, { si L=NIL, fin;
                                  { sinon { si L≠NIL, {I:=I+1;
                                          {           {R:=L;
                                          {           {aller à ca;
               R:=BLINK(RS);
               si R≠NIL, {RS:=R;
                         {aller à c;
               SØ:=NIL; I:=N; [si le symbole n'est pas trouvé,
                                 création de sa p-liste.]
               effectuer N fois {EXFREE(BUF(I),SØ,SØ);
                                {I:=I-1;
              EXFREE(SØ,NIL,SØ);
               EXFREE(SØ,NIL,BLINK(RS)); [mettre le symbole
                                           dans la liste d'accès.]
               fin;
```

§ 4. Le sous-programme "LIRNBR"

effectue une action similaire à "LIRATØM", pour les nombres.

§ 5. Le sous-programme "LIREXPR"

lit une S-expression qui n'est pas un atome et crée sa représentation en mémoire sous forme d'arbre binaire. Après exécution de ce sous-programme, le pointeur QØ contient l'adresse de la représentation.

Comme les règles de construction des S-expressions sont récursives, l'algorithme de traduction des S-expressions en arbres binaires utilise une pile-liste dans laquelle on stocke les noeuds de l'arbre binaire qui sont ancêtres du noeud qu'on est en train de créer et dont le contenu doit encore être complété par la lecture du reste de la S-expression.

Le pointeur STACK donne l'adresse du premier élément de la pile-liste. La variable binaire ETAT vaut "0" si le dernier symbole lu est "(" et "1" ,sinon.
On rajoute un élément à la pile-liste chaque fois qu'on rencontre un symbole "(" et on lui enlève un élément chaque fois qu'on rencontre ")" . Si la pile-liste est vide, la lecture de la S-expression et sa traduction sous forme d'arbre binaire sont terminées.

```
sous-programme LIREXPR;
               K:=80; STACK:=NIL;
a :            LIRSYMB;
               si S="(", { EXFREE(NIL,NIL,QØ); P:=QØ;
                         { EXFREE(NIL,STACK,STACK); ETAT:=0;
               sinon { K:=K+1;
                     { aller à a;
b :            K:=K+1; LIRSYMB;
               si S="(" et ETAT=0, { EXFREE(NIL,NIL,R);
                                   { ALINK(P):=R;
                                   { EXFREE(P,STACK,STACK);
                                   { P:=R; aller à b;
               si S="(" et ETAT=1, { EXFREE(NIL,NIL,R);
                                   { BLINK(P):=R;
                                   { EXFREE(R,STACK,STACK);
                                   { EXFREE(NIL,NIL,P);
                                   { ALINK(R):=P; ETAT:=0;
                                   { aller à b;
               si "A"≤S≤"Z" et ETAT=0, { LIRATØM;
                                       { ALINK(P):=SØ; ETAT:=1;
                                       { aller à b;
               si "A"≤S≤"Z" et ETAT=1, { LIRATØM;
                                       { EXFREE(SØ,NIL,R);
                                       { BLINK(P):=R; P:=R;
                                       { aller à b;
               si "0"≤S≤"9" et ETAT=0, { LIRNBR;
                                       { ALINK(P):=SØ; ETAT:=1;
                                       { aller à b;
               si "0"≤S≤"9" et ETAT=1, { LIRNBR;
                                       { EXFREE(SØ,NIL,R);
                                       { BLINK(P):=R; P:=R;
                                       { aller à b;
               si S="." et ETAT=0, { imprimer "ERREUR A LA LECTURE"
                                   { QØ:=NIL; fin;
               si S="." et ETAT=1, { ba : K:=K+1; LIRSYMB;
                                   { si "A"≤S≤"Z", { LIRATØM;
                                   {               { BLINK(P):=SØ;
                                   {               { aller à b;
                                   { si "0"≤S≤"9", { LIRNBR;
                                   {               { BLINK(P):=SØ;
                                   {               { aller à b;
                                   { si S="(", { EXFREE(NIL,NIL,R);
                                   {           { BLINK(P):=R; P:=R;
                                   {           { EXFREE(P,STACK,
                                   {           {          STACK);
                                   {           { ETAT:=0; aller à b;
                                   { aller à ba;
```

```
si S=")" et ETAT=0, {PØP1(P,STACK);
                     si P=NIL, {QØ:=NIL;
                                fin;
                     ALINK(P):=NIL; ETAT:=1;
                     aller à b;
si S=")" et ETAT=1, {PØP1(P,STACK);
                     si P=NIL, fin;
                     aller à b;
aller à b;

sous-programme PØP1(X,Y); [les arguments sont passés par référence.
               X:=ALINK(Y);
               Y:=BLINK(Y);
               fin;
```

Chapitre 3 : Le programme d'impression.

Ce programme qui sert principalement à imprimer les résultats fournis par "evalquote" imprime la S-expression équivalente à un arbre binaire situé en mémoire. Les sous-programmes utilisés pour réaliser cette fonction sont décrits dans les paragraphes suivants.

§ 1. Le sous-programme "EDIT"

a pour effet d'écrire sur l'imprimante le contenu du buffer BUF2 de 132 caractères.

§ 2. Le sous-programme "PRINTATØM"

met dans le buffer BUF2 la suite de caractères de la représentation externe d'un symbole atomique dont l'adresse est contenue dans le pointeur P . On a :

```
sous-programme PRINTATØM;
               si K>100, {EDIT;
                          K:=2;
               Q:=ALINK(P);
a :            si Q≠NIL, {SUBSTR(BUF2,K,2):=INFØ(Q);
                          Q:=BLINK(Q); K:=K+2; aller à a;
               K:=K+1; fin;
```

§ 3. Le sous-programme "PRINTNBR"

effectue, pour les nombres, un traitement analogue à celui de "PRINTATØM" pour les autres atomes.

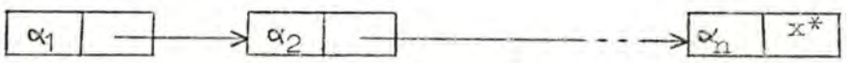
§ 4. Le sous-programme "PRINTCHAR2"

met deux caractères dans BUF2.

§ 5. Le sous-programme "PRINTEXPR"

imprime la S-expression équivalente à l'arbre binaire dont l'adresse se trouve dans le pointeur PØ .
Ce sous-programme utilise deux piles-listes accessibles par les pointeurs STACK et STAET. La première pile-liste mémorise les adresses des différents noeuds de l'arbre binaire, ancêtres de celui qu'on est en train de traiter et dont on n'a pas terminé l'examen; la seconde permet de savoir sous quelle forme on va imprimer les différentes S-expressions correspondant aux sous-arbres binaires restant à explorer.

Pour mettre en évidence la raison d'être de cette seconde pile-liste, considérons la structure suivante:



où x* est un symbole atomique.
Si x* est le symbole "NIL", alors cette structure doit être imprimée sous la forme

$$(\alpha_1 , \ldots , \alpha_n);$$

par contre, si x* est différent de "NIL", elle doit être imprimée sous la forme

$$(\alpha_1.(\alpha_2.(\ldots (\alpha_n.x^*) \ldots))) .$$

Il faut donc parcourir une première fois toute la structure, pour déterminer si c'est une liste ou une paire pointée, avant de commencer à l'imprimer. La seconde pile-liste est utilisée pour mémoriser le résultat de ce test pour toutes les listes et toutes les paires pointées en cours d'impression.

Si l'on avait décidé au départ, que les seules S-expressions prises en considération par le LISP seraient des atomes ou des listes, on aurait pu écrire un algorithme beaucoup plus simple. En particulier, une seule pile-liste suffit.
Nous donnons d'abord, un programme simplifié qui ne traite que des listes.

```
sous-programme PRINTLIST;
               K:=2; STACK:=NIL; R:=PØ;
               PRINTCHAR2("( ");
a :            P:=ALINK(R); Q:=ALINK(P);
               PP:=ALINK(Q); R:=BLINK(R);
               si PP<0, { si BLINK(P)=NUMBER, PRINTNBR;
                        { sinon PRINTATØM;
                        { b :
                        { si R=NIL, { PRINTCHAR2(") ");
                        {           { si STACK=NIL, { EDIT;
                        {           {               { fin;
                        {           { PØP1(R,STACK);
                        {           { aller à b;
```

```
                sinon {PRINTCHAR2("( ");
                       EXFREE(R,STACK,STACK); R:=P;
                aller à a;
```

Voici, maintenant l'algorithme général pour des S-expressions quelconques.

```
sous-programme PRINTEXPR;
               K:=80;STACK,STAET:=NIL; P:=PØ;
a :            Q:=ALINK(P); PP:=ALINK(Q);
               si PP<0, {si BLINK(P)=NUMBER, PRINTNBR; [P pointe
                                                  sur un atome.]
                         {sinon PRINTATØM;
                         si STAET=NIL, aller à c;
                         si STACK≠NIL, PØP1(P,STACK);
                         aller à b;
               PRINTCHAR2("( ");Q:=P;
aa :           Q:=BLINK(Q); [phase d'exploration.]
               si Q=NIL, EXFREE(2,STAET,STAET); [P pointe sur
                                                 une liste.]
               sinon {PP:=ALINK(Q);
                      si ALINK(PP)<0, EXFREE(1,STAET,STAET);
                                                [P pointe sur
                                                 une paire pointée.
                      sinon aller à aa;
b :            ETAT:=ALINK(STAET);
               aller à switch(ETAT);
switch(1) :    EXFREE(BLINK(P),STACK,STACK); [exploration sous-
                                               arbre gauche.]
               P:=ALINK(P); ALINK(STAET):=3;
               aller à a;
switch(2) :    si P=NIL, {PRINTCHAR2(") "); [exploration liste.]
                          si STACK=NIL, aller à c;
                          STAET:=BLINK(STAET); PØP1(P,STACK);
                          aller à b;
               EXFREE(BLINK(P),STACK,STACK);
               P:=ALINK(P); aller à a;
switch(3) :    PRINTCHAR2(". "); [imprimer "."]
               ALINK(STAET):=4; aller à a;
switch(4) :    PRINTCHAR2(") "); [fin exploration paire pointée.]
               STAET:=BLINK(STAET);
               si STAET≠NIL, aller à b;
c :            EDIT;
               fin;
```

La fonction "PRINT"
---

correspond à la possibilité d'utiliser explicitement le sous-programme "PRINTEXPR" dans un programme LISP. Elle admet un argument qui est la S-expression à imprimer.

Chapitre 4 : Le Superviseur.

Lorsqu'on veut traiter un problème en LISP, on écrit généralement plusieurs programmes dont les exécutions successives permettent d'obtenir le résultat. On peut aussi désirer exécuter plusieurs traitements en un seul passage machine. On a donc créé un programme appelé "Superviseur" chargé d'enchaîner les traitements des programmes LISP successifs constitutifs d'un ou plusieurs traitements.
En outre, on a la possibilité de donner des ordres à ce programme superviseur permettant principalement deux choses.

- Gérer la mémoire contenant les représentations internes des S-expressions.

- Gérer la mise en page des résultats.

Ces ordres sont fournis au superviseur par l'intermédiaire de "cartes contrôle" insérées dans le deck de cartes des programmes LISP.
Lorsqu'on veut exécuter un ou plusieurs traitements, on doit donc constituer un deck contenant deux types de cartes:

- cartes contrôle: ordres pour le superviseur,

- cartes programme: texte des programmes LISP à évaluer.

Voici la liste des cartes contrôle disponibles dans notre système et leur action.

- ~~bb~~DEBUT,LISP : provoque l'initialisation de la mémoire: création de la liste libre. C'est la première carte du deck.

- ~~bbbb~~ : (carte blanche) signale que les cartes qui suivent constituent un programme LISP. Un programme doit toujours être précédé d'au moins une carte de ce type.

- ~~bb~~EJECT : la présence de cette carte provoque un saut de page à l'imprimante.

- ~~bb~~RECLAIM,LISP : cette carte provoque une réorganisation de la mémoire: on appelle le "garbage collector". (voir quatrième partie.) Les p-listes de tous les atomes lus précédemment sont préservées ainsi que les listes d'accès. Toutes les autres cellules sont rangées dans une nouvelle liste libre. L'emploi judicieux de cette carte peut, dans certains cas, faire éviter l'appel du garbage collector dans les programmes qui suivent et comme le nombre de cellules à marquer est moins grand, faire gagner du temps d'exécution.

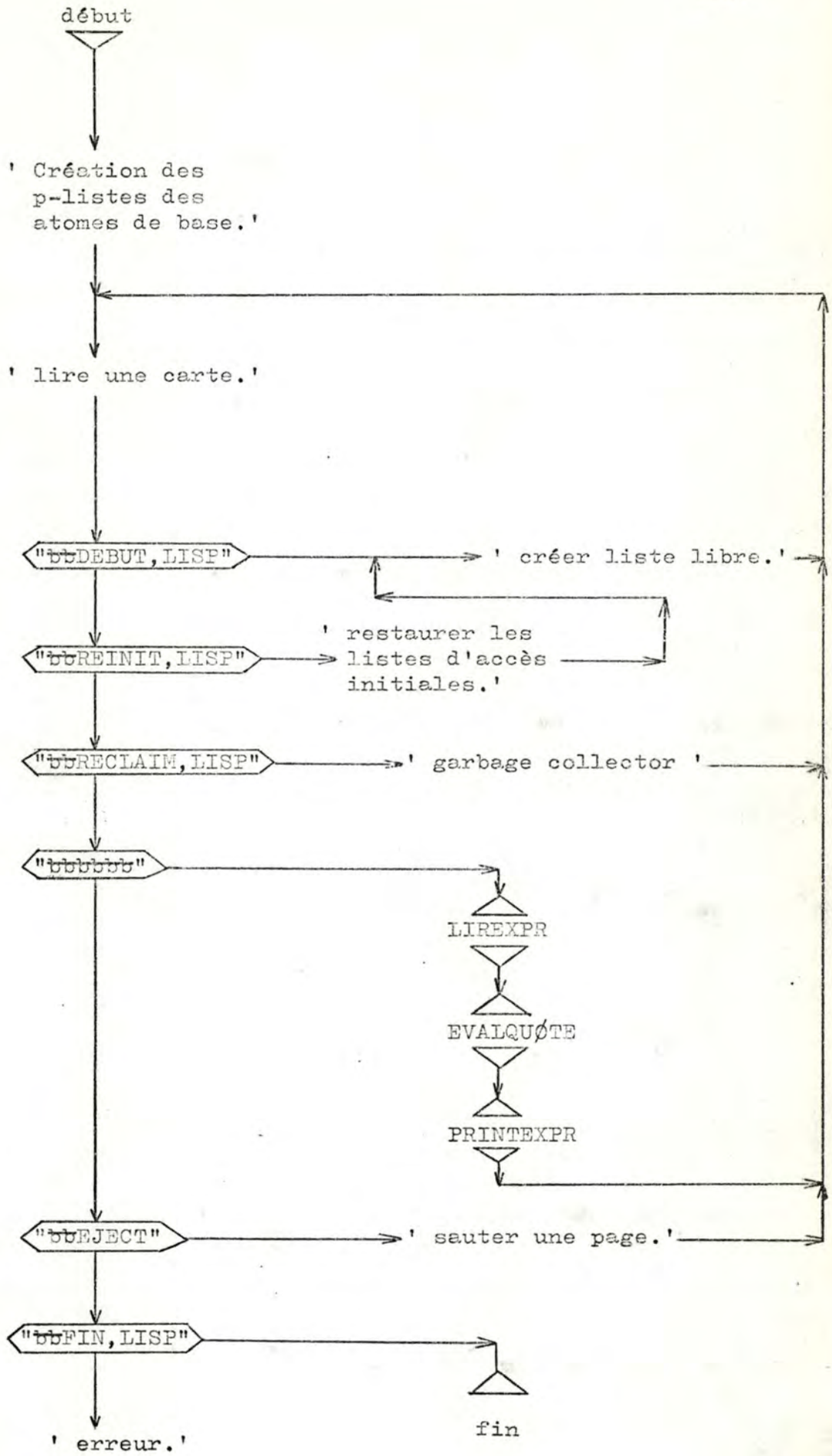
- ~~bb~~REINIT,LISP : cette carte provoque la mise dans la liste libre de toutes les cellules utilisées depuis le début de l'exécution. Seules les listes présentes à l'initialisation de la mémoire sont préservées.
Cette carte permet de récupérer le maximum de mémoire. Toutefois, son emploi est délicat: si on a modifié, précédemment,la p-liste d'un atome de base, par exemple, par le programme

(CSET,NIL,NIL) ,

il faudra prendre des précautions, car cette p-liste sera détériorée. On peut, par exemple, après la carte ~~bb~~REINIT,LISP, placer immédiatement des programmes "CSET" ou "DEFINE" pour tous ces atomes.

- ~~bb~~FIN,LISP : cette carte est la dernière du deck, elle provoque l'arrêt du programme superviseur.

Le fonctionnement du superviseur est décrit par l'organigramme de la page III.11.



N.B.

Un test vérifié provoque un branchement vers la droite,un test non vérifié provoque un branchement vers le bas.

QUATRIEME PARTIE : LE GARBAGE CØLLECTØR.

Chapitre 1 : Le problème de la récupération des zones inutiles.

Nous avons supposé jusqu'à présent que la mémoire qui se trouvait à notre disposition était illimitée. En effet, chaque fois qu'un traitement nécessitait l'utilisation d'une nouvelle cellule de mémoire, nous appelions le sous-programme "EXFREE" qui fournissait la cellule désirée, en supposant que c'était toujours possible. Or, la mémoire à notre disposition est au maximum de 32 K et certains traitements nécessitent l'obtention de plus de 32.000 cellules, l'interpréteur étant très récursif. Toutefois, si un traitement nécessite l'obtention de plus de 32.000 cellules, cela ne veut pas dire que ces cellules sont toutes actives à un moment donné de l'exécution. (Auquel cas, la poursuite du traitement est impossible.) Au contraire, généralement, un grand nombre de ces cellules contiennent des renseignements périmés et on peut donc les récupérer pour effectuer d'autres tâches.
Pour s'en convaincre, on peut examiner l'exemple suivant.

Considérons l'évaluation de la forme

$$(\varphi^*, \varepsilon_1^*, \ \ldots \ , \varepsilon_n^*)$$

par "eval": ce sous-programme va se rappeler n fois, lui-même, pour calculer les n formes $\varepsilon_1^*, \ \ldots \ , \varepsilon_n^*$. Le traitement de ces n formes provoquera l'addition de nouvelles cellules à la pile-liste et à la a-liste. Une fois que les valeurs $\alpha_1, \ \ldots \ , \alpha_n$ des n formes sont calculées, la pile-liste et la a-liste sont restaurées à leur valeur initiale et toutes les cellules utilisées pour leur évolution entre le début et la fin de l'évaluation de $\varepsilon_1^*, \ \ldots \ , \varepsilon_n^*$ sont non seulement inutiles mais même inaccessibles.

On peut imaginer deux méthodes pour effectuer la récupération de mémoire.

- Réintégrer les cellules à la liste libre au moment même où elles deviennent inutiles.

- Récupérer toutes les cellules inutiles au moment où la liste libre est vide.

La première méthode est souvent utilisée pour la gestion de la mémoire dans des langages tels que Algol 60, mais il faut remarquer que, dans ce cas, les zones de mémoire activées en dernier lieu sont les premières à être libérées et qu'en conséquence, la récupération des zones inutiles ne nécessite pratiquement aucun traitement; seule, la mise à jour d'un pointeur est, éventuellement, nécessaire. Il en va tout autrement en LISP où l'on crée des structures de formes très variées dont certaines parties peuvent devenir inutiles et cela de façon tout à fait imprévisible.

En LISP, la seconde méthode offre les avantages suivants:

- le seul évènement susceptible de provoquer la récupération est l'absence de cellule dans la liste libre. Cet évènement peut se produire n'importe quand, lors de l'évaluation, mais sera toujours détecté à l'intérieur du sous-programme "EXFREE" , on ne doit donc pas se préoccuper de savoir quelles cellules deviennent inutiles à un moment donné, ni programmer explicitement leur réintégration à la liste libre ce qui nécessiterait la création d'un sous-programme "RETURNFREE". Ce programme devrait être appelé un peu partout dans l'interpréteur; de plus, et ceci est très important, il n'est pas toujours simple d'établir le moment ou des cellules deviennent inaccessibles. En LISP, c'est pratiquement impossible dans bien des cas. C'est ce dernier point qui est décisif dans le choix de la méthode.

- Si le traitement nécessite l'obtention de moins de 32.000 cellules de mémoire, la récupération des cellules est inutile et on gagne du temps.

On utilisera donc toujours la seconde méthode, en LISP.

L'algorithme de récupération procède en deux phases.

(1) "Marquage" de toutes les cellules actives, par parcourt de tous les arbres binaires accessibles à partir de certaines adresses de base judicieusement choisies.

(2) Parcourt séquentiel de la mémoire durant lequel on effectue deux choses:

- constitution d'une nouvelle liste libre avec les cellules non marquées,

- "démarquage" des cellules actives.

Avant de décrire en détail, au chapitre 2, le "garbage collector" (c'est à dire: programme récupérateur.) utilisé dans notre système, nous donnons une description sommaire d'une méthode existant dans beaucoup de systèmes LISP et basée sur une organisation différente de la mémoire.

Supposons qu'au lieu d'une pile-liste, on utilise un stack linéaire placé à une extrémité de la mémoire, tandis que la zone contenant les S-expressions est placée à l'autre extrémité de la mémoire. (Voir figure IV.1)
Entre ces deux zones, on a une zone de mémoire tenant lieu de liste libre bien que dépourvue de toute structure de liste. Deux pointeurs PTI et PTS délimitent cette zone. L'extraction d'une cellule pour le stack correspond à l'opération

PTS:=PTS-1;

l'extraction d'une cellule pour la construction des S-expressions correspond à

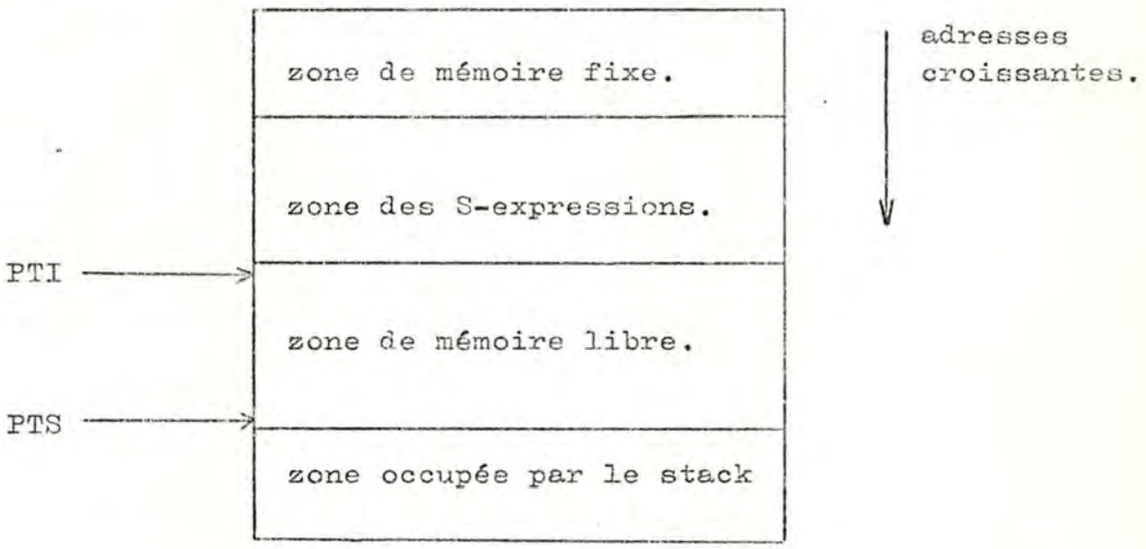
PTI:=PTI+1;

l'opération

PTS:=PTS+1;

correspond à la suppression d'un niveau au stack.
On voit que la récupération des cellules inutiles du stack peut se faire suivant la première méthode préconisée plus haut, cela est dû à sa structure "last-in,first-out" .

figure IV.1

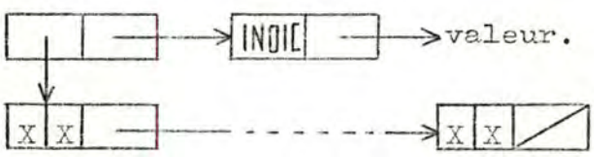


Par contre, la récupération des cellules constitutives des S-expressions ne peut se faire que par utilisation d'un garbage collector. Celui-ci est activé lorsque PTI = PTS , c'est à dire lorsque la zone libre est vide. Il doit compacter toutes les S-expressions actives au début de la mémoire car la zone libre qu'il doit restituer doit être formée de cellules contiguës. Cette opération est assez délicate, car si on recopie une S-expression au début de la mémoire, il faut aussi modifier toutes les références à celle-ci faites dans les autres S-expressions et dans l'interpréteur.
Un tel garbage collector est appelé "à compactage". Son écriture est plus compliquée que celle du garbage collector employé dans notre système. Son exécution est aussi plus lente. Cependant, il faut remarquer que l'utilisation d'un stack linéaire est moins coûteuse en temps et en place que celle d'une pile-liste. Nous avons choisi une gestion par pile-liste, parce que, étant plus homogène, elle permet une écriture plus aisée des programmes.
Pour plus de renseignements sur un garbage collector LISP à compactage, on pourra consulter, par exemple, la référence [6].

Enfin, avant de décrire notre programme récupérateur, disons encore que le problème est fort simplifié en LISP du fait que les cellules de mémoire utilisées ont une structure simple et rigide: deux pointeurs et que l'on peut aisément déterminer un petit nombre d'adresses de base permettant d'accéder à toutes les cellules actives. Le problème général de la récupération lorsque les cellules chaînées ont une structure tout à fait quelconque, est beaucoup plus compliqué. A ce sujet, on pourra consulter, par exemple, la référence [9].

Chapitre 2 : Le garbage collector et le sous-programme "EXFREE".

La p-liste d'un atome a la structure suivante:



Lors du marquage d'un atome par le garbage collector, il est clair qu'il faut marquer toute sa p-liste. Mais nous allons définir un sous-programme "MARKATØM" qui marque seulement la liste contenant les caractères de la représentation externe de l'atome, le reste de la p-liste étant marqué par le sous-programme général de marquage des S-expressions: "MARKTREE".
Remarquons que la p-liste d'un atome ne présente pas toujours une structure d'arbre binaire et peut comporter des cycles. Pour cette raison, "MARKTREE" doit toujours marquer une cellule avant de marquer ses descendants, sinon il risquerait de boucler; ainsi, un algorithme de marquage en "endorder" risquerait d'échouer. Il faut un algorithme du type "preorder".

Le marquage d'une cellule consiste a inverser le signe de l'adresse contenue dans le champ BLINK de la cellule:

si α est l'adresse d'une cellule et
si BLINK $(\alpha)<0$,

la cellule est considérée comme marquée.
Rappelons que si le test

$$\text{ALINK}\ (\alpha)<0$$

est positif, cela veut dire que la cellule contient des caractères dans son champ INFØ.

Le sous-programme "MARKTREE" marque une structure dont l'adresse est contenue dans le pointeur PPØ. C'est une combinaison de deux algorithmes.

Le premier utilise un stack de dimension maximum réduite, destiné a contenir les adresses des structures restant à marquer.

Si ce stack est plein, on utilise un second algorithme, plus compliqué, qui n'emploie pas de stack ou, plutôt, qui constitue son stack à l'intérieur même de la structure à marquer en la modifiant temporairement.
Ces deux algorithmes sont du type "preorder" pour la raison exposée plus haut.
Voici le texte des sous-programmes "MARKATØM" et "MARKTREE".

```
sous-programme MARKATØM;  [Q pointe vers la liste à marquer.]
a :            PPQ:=BLINK(Q);
               BLINK(Q):=-PPQ;
               si PPQ=NIL, fin;
               Q:=PPQ;
               aller à a;


sous-programme MARKTREE;
               PS:=0; PR:=PPØ;
               PT:=NUL; [NUL contient une configuration
                           de bits non utilisée ailleurs.]
a :            PZ:=BLINK(PR); PZZ:=ALINK(PR);
               si PZ<0, aller à b; [cellule déjà marquée.]
               si PZZ<0, {Q:=PR; [liste de caractères.]
                         {MARKATØM; aller à b;
               BLINK(PR):=-PZ; [marquage cellule.]
               si PS<N, {PS:=PS+1; [sauver sous-arbre droit.]
                        {PSTACK(PS):=PZ;
                        {PR:=PZZ; [marquer sous-arbre gauche.]
                        {aller à a;
               PP:=PR; aller à c; [stack plein.]
b :            si PS=0, fin;
               PR:=PSTACK(PS); [marquer un nouvel arbre.]
               PS:=PS-1;
               aller à a;
c :            [second algorithme.]
               PQ:=ALINK(PP); [marquer sous-arbre gauche.]
               PZ:=BLINK(PQ);
               si PZ<0, aller à d;
               si ALINK(PQ)<0, {Q:=PQ;
                               {MARKATØM; aller à d;
               BLINK(PQ):=-PZ;ALINK(PP):=-PT;
               PT:=PP; PP:=PQ; aller à c;
d :            PQ:=-BLINK(PP); [marquer sous-arbre droit.]
               PZ:=BLINK(PQ);
               si PZ<0, aller à e;
               BLINK(PQ):=-PZ; BLINK(PP):=-PT;
               PT:=PP; PP:=PQ; aller à c;
```

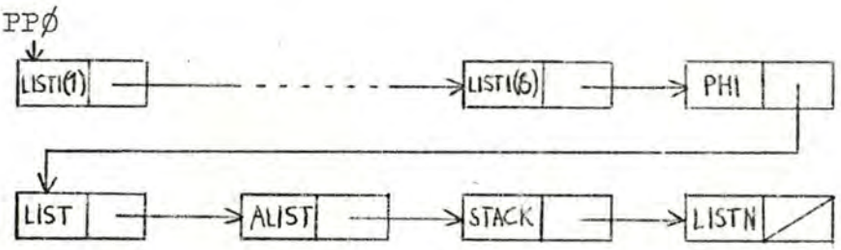
```
e :     si PT=NUL, aller à b; [retour au premier
                                algorithme.]
        PQ:=PT; PZ:=ALINK(PQ);
        si PZ<0, {PT:=-PZ; ALINK(PQ):=PP;
                 {PP:=PQ; aller à d;
        PT:=-BLINK(PQ); BLINK(PQ):=-PP;
        PP:=PQ; aller à e;
```

On a indiqué, ci-dessous, les pointeurs donnant accès à toutes les cellules actives, suivant l'endroit où il est fait appel au garbage collector.

- au cours de la lecture : QØ,SØ,STACK,ACCU.
- au cours de l'impression : PØ,STAET,STACK,ACCU.
- au cours de "apply" : PHI,LIST,ALIST,STACK.
- au cours de "eval" : EPSI,ALIST,STACK,ACCU.

A ces pointeurs, il faut rajouter les six pointeurs LISTI et le pointeur LISTN permettant de sauvegarder toutes les p-listes et leur listes d'accès. Ces sept pointeurs sont rangés en permanence dans une liste fixe située dans la zone de mémoire réservée et dont l'adresse est contenue dans le pointeur PPØ.
Au moment de l'appel du garbage collector, la valeur 1 , 2 , 3 , 4 de l'indicateur INDIC permet de savoir dans laquelle des quatre situations ci-dessus, on se trouve. On range alors, dans la liste fixe, les adresses convenables, ensuite on appelle le sous-programme "MARKTREE" qui marque toutes les cellules actives; après cela, on reconstitue une liste libre et on peut retourner à l'endroit de l'appel du garbage collector.

Exemple. Si INDIC=3 , avant l'appel de "MARKTREE" , on a la situation ci-dessous.



Nous pouvons maintenant écrire le sous-programme "EXFREE" d'obtention d'une nouvelle cellule.

```
sous-programme EXFREE(X,Y,Z);[ les arguments sont passés
                                        par référence.]
                si FREE=NIL, aller à switch(INDIC);
a :             Q:=FREE; FREE:=BLINK(Q);
                ALINK(Q):=X; BLINK(Q):=Y; Z:=Q;
                fin;
switch(1) :     ALINK(PPØ+6):=QØ; ALINK(PPØ+7):=SØ;[ lecture.]
                ALINK(PPØ+8):=STACK; ALINK(PPØ+9):=ACCU; aller à b;
switch(2) :     ALINK(PPØ+6):=PØ; ALINK(PPØ+7):=STACK;[ impression.]
                ALINK(PPØ+8):=STAET; ALINK(PPØ+9):=ACCU; aller à b;
switch(3) :     ALINK(PPØ+6):=PHI; ALINK(PPØ+7):=STACK; ["apply".]
                ALINK(PPØ+8):=LIST; ALINK(PPØ+9):=ALIST; aller à b;
switch(4) :     ALINK(PPØ+6):=EPSI; ALINK(PPØ+7):=STACK;[ "eval".]
                ALINK(PPØ+8):=ACCU; ALINK(PPØ+9):=ALIST;
b :             MARKTREE; [ marquage des cellules actives.]
                imprimer "GARBAGE COLLECTOR";
                PI:=MAXFREE;[ recherche dernière cellule libre.]
c :             PK:=BLINK(PI);
                si PK<0, { BLINK(PI):=-PK;
                           si PI=MINFREE, { imprimer "MEMØIRE
                                                INSUFFISANTE";
                                            fin anormale;
                           PI:=PI-1; aller à c;
                BLINK(PI):=NIL;
                FREE:=MINFREE;[ recherche première cellule libre.]
d :             PJ:=BLINK(FREE);
                si PJ<0, { BLINK(FREE):=-PJ;
                           FREE:=FREE+1; aller à d;
                PK:=FREE; [ constitution liste libre.]
                PL:=PK;
e :             si PL>PI, aller à a;
                PJ:=BLINK(PK);
                si PJ<0, BLINK(PK):=-PJ;
                sinon { BLINK(PL):=PK;
                        PL:=PK;
                PK:=PK+1; aller à e;
```

CINQUIEME PARTIE : DEUX EXEMPLES.

Premier exemple : L'algorithme de Wang.

Cet exemple est un grand classique du LISP puisqu'il figurait déjà dans la référence [2] parue peu après la création du LISP. Nous ne décrirons donc pas la méthode utilisée, nous contentant de donner les indications nécessaires à la compréhension des exemples traités.

Etant donné deux ensembles de termes de la logique des propositions, l'algorithme de Wang détermine si on peut déduire le second ensemble du premier.

Les opérateurs de la logique des propositions sont représentés par les symboles atomiques "AND" , "ØR" , "NØT" , "IMPLIES" , "EQUIV" ; les termes élémentaires sont représentés par d'autres symboles atomiques et les termes composés sont construits, sous forme préfixée, par composition des opérateurs et de termes plus simples.

Exemples.

(ØR,A,B)	désigne le terme	$A \vee B$,
(AND,A,B)	" " "	$A \wedge B$,
(NØT,A)	" " "	$\neg A$,
(IMPLIES,A,B)	" " "	$A \Rightarrow B$,
(EQUIV,A,B)	" " "	$A \Leftrightarrow B$,

Convenons, alors, de désigner par $\varepsilon_1^*, \ldots, \varepsilon_m^*$ et $\omega_1^*, \ldots, \omega_n^*$ deux ensembles de termes de la logique des propositions écrits dans la notation indiquée ci-dessus.

La fonction "THEØREM" ,

écrite en mode récursif uniquement, délivre le résultat "T" si on peut déduire les termes $\omega_1^*, \ldots, \omega_n^*$ des termes $\varepsilon_1^*, \ldots, \varepsilon_m^*$, et le résultat "F" , sinon.

Le programme à présenter à l'interpréteur est:

(THEØREM(ARRØW($\varepsilon_1^*, \ldots, \varepsilon_m^*$)($\omega_1^*, \ldots, \omega_n^*$))) .

```
(DEFINE(
(THEOREM(LAMBDA(S)(TH1 NIL NIL(CADR S)(CADDR S))))

(TH1(LAMBDA(A1 A2 A C)(COND((NULL A)
 (TH2 A1 A2 NIL NIL C))(T
(OR(MEMBER(CAR A)C)(COND((ATOM(CAR A))
 (TH1(COND((MEMBER(CAR A)A1)A1)
 (T(CONS(CAR A)A1)))A2(CDR A)C))
 (T(TH1 A1(COND((MEMBER(CAR A)A2)A2)
 (T(CONS(CAR A)A2)))(CDR A)C)))))))))

 (TH2(LAMBDA(A1 A2 C1 C2 C)(COND
 ((NULL C)(TH A1 A2 C1 C2))
 ((ATOM(CAR C))(TH2 A1 A2(COND
 ((MEMBER(CAR C)C1)C1)(T
 (CONS(CAR C)C1)))C2(CDR C)))
 (T(TH2 A1 A2 C1(COND((MEMBER
 (CAR C)C2)C2)(T(CONS(CAR C)C2)))
 (CDR C))))))

 (THL(LAMBDA(U A1 A2 C1 C2)(COND
 ((EQ(CAR U)(QUOTE NOT))(TH1R(CADR U)A1 A2 C1 C2))
 ((EQ(CAR U)(QUOTE AND))(TH2L(CDR U)A1 A2 C1 C2))
 ((EQ(CAR U)(QUOTE OR))(AND(TH1L(CADR U)A1 A2 C1 C2)
 (TH1L(CADDR  U)A1 A2 C1 C2)))
 ((EQ(CAR U)(QUOTE IMPLIES))(AND(TH1L(CADDR U)  A1 A2 C1 C2)
 (TH1R(CADR U)A1 A2 C1 C2)))
 ((EQ(CAR U)(QUOTE EQUIV))(AND(TH2L(CDR U)A1 A2 C1 C2)
 (TH2R(CDR U)A1 A2 C1 C2)))
 (T(PRINT(LIST(QUOTE THL) U A1 A2 C1 C2)))
 )))
 (TH1R(LAMBDA(V A1 A2 C1 C2)(COND,
 ((ATOM V)(OR(MEMBER V A1)
 (TH A1 A2(CONS V C1)C2)))
 (T(OR(MEMBER V A2)(TH A1 A2 C1(CONS V C2))))
 )))
 (TH2L(LAMBDA(V A1 A2 C1 C2)(COND
 ((ATOM (CAR V))(OR(MEMBER(CAR V)C1)
 (TH1L(CADR V)(CONS(CAR V)A1)A2 C1 C2)))
 (T(OR(MEMBER(CAR V)C2)(TH1L(CADR V)A1(CONS(CAR V)
                    A2) C1 C2)))
 )))
 (TH1L(LAMBDA(V A1 A2 C1 C2)(COND
 ((ATOM V)(OR(MEMBER V C1)
 (TH(CONS V A1) A2 C1 C2)))
 (T(OR(MEMBER V C2)(TH A1(CONS V A2)C1 C2)))
 )))
 ))
 DEBUT DE EVALQUOTE
 FIN DE EVALQUOTE
NIL
 EJECT
```

```
  (DEFINE(
  (TH(LAMBDA(A1 A2 C1 C2)(COND((NULL A2)(COND((NULL C2)F) (T
  (THR(CAR C2)A1 A2 C1(CDR C2)))))(T(THL(CAR A2)A1(CDR A2)
          C1 C2)))))
  (THR(LAMBDA(U A1 A2 C1 C2)(COND
  ((EQ(CAR U)(QUOTE NOT))(TH1L(CADR U)A1 A2 C1 C2))
  ((EQ(CAR U)(QUOTE AND))(AND(TH1R(CADR U)A1 A2 C1 C2)
  (TH1R(CADDR U)A1 A2 C1 C2)))
  ((EQ(CAR U)(QUOTE OR))(TH2R(CDR U)A1 A2 C1 C2))
  ((EQ(CAR U)(QUOTE IMPLIES))(TH11(CADR U)(CADDR U)
                        A1 A2 C1 C2))
  ((EQ(CAR U)(QUOTE EQUIV))(AND(TH11(CADR U)(CADDR U)
  A1 A2 C1 C2)(TH11(CADDR U)(CADR U)A1 A2 C1 C2)))
  (T(PRINT(LIST(QUOTE THR)U A1 A2 C1 C2)))
  )))
  (TH2R(LAMBDA(V A1 A2 C1 C2)(COND
  ((ATOM(CAR V))(OR(MEMBER(CAR  V)A1)
  (TH1R(CADR V)A1 A2(CONS(CAR V)C1)C2)))
  (T(OR(MEMBER(CAR V)A2)(TH1R(CADR V)A1 A2 C1
  (CONS(CAR V)C2))))
  )))
  (TH11(LAMBDA(V1 V2 A1 A2 C1 C2)(COND
  ((ATOM V1)(OR(MEMBER V1 C1)(TH1R V2(CONS V1 A1)A2 C1 C2)))
  (T(OR(MEMBER V1 C2)(TH1R V2 A1(CONS V1 A2)C1 C2)))
  )))
    ))
  DEBUT DE EVALQUOTE
  FIN DE EVALQUOTE
NIL
  EJECT
```

```
  (THECREM
  (ARROW((AND(NOT P)(NOT Q)))((EQUIV P Q))))
  DEBUT DE EVALQUOTE
  FIN DE EVALQUOTE
T

  (THECREM
  (ARROW((IMPLIES P Q))((IMPLIES Q P))))
  DEBUT DE EVALQUOTE
  FIN DE EVALQUOTE
F

  (THECREM
  (ARROW((OR A (NOT B)))(( IMPLIES (AND P Q)(EQUIV P Q)))))
  DEBUT DE EVALQUOTE
  FIN DE EVALQUOTE
T

 (THEOREM
 (ARROW
 ( Q,(IMPLIES P Q),(IMPLIES Q R),(IMPLIES R S),(IMPLIES S,P),(NOT A))
 ((AND P (AND Q (AND R S))),(IMPLIES A Q))
 ))
  DEBUT DE EVALQUOTE
  FIN DE EVALQUOTE
T
  EJECT
```

La fonction "WANG",

écrite sur la base d'une forme de type PRØG, est tirée de la référence [7]. Elle délivre le même résultat que "THEØREM" et imprime, en plus, des résultats intermédiaires.

Le programme doit être présenté à l'interpréteur sous la forme:

(WANG(NIL,ARRØW(
(IMPLIES
(AND,ε_1^*(AND,ε_2^*(... (AND,ε_{m-1}^*,ε_m^*) ...)))
(AND,ω_1^*(AND,ω_2^*(... (AND,ω_{n-1}^*,ω_n^*) ...)))
)))) .

```
(DEFINE(
(WANG(LAMBDA(X)(HAO(CAR X)(CADDR X))))
(AUXI(LAMBDA(A B C D)(WANG(PRINT(LIST(APPEND A B)
    (QUOTE ARROW)(APPEND C D))))))
(TAIL(LAMBDA(X)(COND((ATOM(CAR X))(TAIL(CDR X)))(T(CDR X)))))
(HEAD(LAMBDA(X)(COND((ATOM(CAR X))(CONS(CAR X)
    (HEAD(CDR X))))(T NIL))))
(W(LAMBDA(X)(COND((ATOM(CAR X))(W(CDR X)))(T(CAR X)))))
(COMMON(LAMBDA(X Y)(COND((NULL X)F  )((MEMBER(CAR X) Y)T)
                    (T(COMMON(CDR X) Y)))))
(ATOMIC(LAMBDA(X)(COND((NULL X)T)
        ((ATOM(CAR X))(ATOMIC(CDR X)))(T F  ))))
(HAO (LAMBDA(X Y)(PROG(S1 S2 C U V)
(COND((ATOMIC X)(GO(QUOTE A)))
      (T NIL))
(SET(QUOTE S1)(HEAD X))
(SET(QUOTE S2)(TAIL X))(SET(QUOTE U)(CDR(W X)))
(SET(QUOTE C)(CAR(W X)))
(COND ((EQ C (QUOTE NOT))(RETURN(AUXI S1 S2 Y U)))
      ((EQ C (QUOTE AND))(RETURN(AUXI S1(APPEND U S2)NIL Y)))
      ((EQ C (QUOTE EQUIV))(RETURN(AND(AUXI U (APPEND S1 S2)
                          NIL Y)(AUXI S1 S2 Y U))))
                (T NIL))
(SET(QUOTE V)(CADR U))
(SET(QUOTE U)(CAR U))
(COND((EQ C(QUOTE OR))(RETURN(AND(AUXI S1 (CONS U S2)NIL Y)
                                 (AUXI S1 (CONS V S2)NIL Y))))
      ((EQ C(QUOTE IMPLIES))(RETURN(AND(AUXI S1 (CONS V S2)
                                NIL Y)(AUXI S1 S2 Y(CONS U NIL)))))
      (T(RETURN(QUOTE ERROR1))))
A       (COND((ATOMIC Y)(RETURN(COMMON X Y))))
(SET(QUOTE S1)(HEAD Y))(SET(QUOTE S2)(TAIL Y))
(SET(QUOTE U)(CDR(W Y)))
(SET(QUOTE C)(CAR(W Y)))
(COND((EQ C(QUOTE NOT))(RETURN(AUXI U X S1 S2)))
    ((EQ C(QUOTE OR))(RETURN(AUXI NIL X S1(APPEND U S2))))
      (T NIL))
(SET(QUOTE V)(CADR U))
(SET(QUOTE U)(CAR U))(
COND ((EQ C(QUOTE AND))(RETURN(AND(AUXI NIL X S1(CONS U S2))
       (AUXI NIL X S1 (CONS V S2)))))
      ((EQ C(QUOTE IMPLIES))(RETURN(AUXI NIL (CONS U X) S1
                          (CONS V S2))))
      ((EQ C(QUOTE EQUIV))(RETURN(AND(AUXI NIL(CONS U X)S1
      (CONS V S2))(AUXI NIL(CONS V X)S1(CONS U S2)))))
         (T(RETURN(QUOTE ERROR2))))))
))
DEBUT DE EVALQUOTE
FIN DE EVALQUOTE
NIL
EJECT
```

```
  (WANG
  (NIL ARROW ((IMPLIES(AND(NOT P)(NOT Q))(EQUIV P Q)))))
  DEBUT DE EVALQUOTE
( ( ( AND  ( NOT  P  ) ( NOT  Q  ) ) ) ARROW  ( ( EQUIV  P  Q  ) ) )
( ( ( NOT  P  ) ( NOT  Q  ) ) ARROW  ( ( EQUIV  P  Q  ) ) )
( ( ( NOT  Q  ) ) ARROW  ( ( EQUIV  P  Q  ) P  ) )
( NIL  ARROW  ( ( EQUIV  P  Q  ) P  Q  ) )
( ( P  ) ARROW  ( Q  P  Q  ) )
( ( Q  ) ARROW  ( P  P  Q  ) )
  FIN DE EVALQUOTE
T

  (WANG
  (NIL ARROW ((IMPLIES P(NOT P)))))
  DEBUT DE EVALQUOTE
( ( P  ) ARROW  ( ( NOT  P  ) ) )
( ( P  P  ) ARROW  NIL  )
  FIN DE EVALQUOTE
F
  EJECT
```

```
(WANG
(NIL ARROW
((IMPLIES
(ANC(IMPLIES P Q)(AND (OR P R)(NOT R)))
(AND P Q)
))))
  DEBUT DE EVALQUOTE
( ( ( AND ( IMPLIES P Q ) ( AND ( OR P R ) ( NOT R ) ) ) ARROW ( ( AND P Q ) ) )
( ( ( IMPLIES P Q ) ( AND ( OR P R ) ( NOT R ) ) ) ARROW ( ( AND P Q ) ) )
( ( Q ( AND ( OR P R ) ( NOT R ) ) ) ARROW ( ( AND P Q ) ) )
( ( Q ( OR P R ) ( NOT R ) ) ARROW ( ( AND P Q ) ) )
( ( Q P ( NOT R ) ) ARROW ( ( AND P Q ) ) )
( ( Q P ) ARROW ( ( AND P Q ) R ) )
( ( Q P ) ARROW ( P R ) )
( ( Q P ) ARROW ( Q R ) )
( ( Q R ( NOT R ) ) ARROW ( ( AND P Q ) ) )
( ( Q R ) ARROW ( ( AND P Q ) R ) )
( ( Q R ) ARROW ( P R ) )
( ( Q R ) ARROW ( Q R ) )
( ( ( AND ( OR P R ) ( NOT R ) ) ) ARROW ( ( AND P Q ) P ) )
( ( ( OR P R ) ( NOT R ) ) ARROW ( ( AND P Q ) P ) )
( ( P ( NOT R ) ) ARROW ( ( AND P Q ) P ) )
( ( P ) ARROW ( ( AND P Q ) P R ) )
( ( P ) ARROW ( P P R ) )
( ( P ) ARROW ( Q P R ) )
( ( R ( NOT R ) ) ARROW ( ( AND P Q ) P ) )
( ( R ) ARROW ( ( AND P Q ) P R ) )
( ( R ) ARROW ( P P R ) )
( ( R ) ARROW ( Q P R ) )
  FIN DE EVALQUOTE
T
  EJECT
```

Second exemple : Un programme de dérivation formelle d'expressions polynomiales.

Nous allons décrire un ensemble de fonctions permettant d'effectuer la dérivation formelle d'expressions polynomiales écrites suivant la syntaxe du FØRTRAN. Cet exemple donne une idée de ce qu'on peut réaliser, du point de vue pratique, avec le LISP.

Le LISP est bien adapté à la dérivation d'expressions arithmétiques si elles sont écrites suivant la syntaxe suivante:

<expression>::=(<opérateur diadique><expression><expression>)
(<opérateur monadique><expression>)
<symbole atomique> .

Toutefois, cette notation, préfixée et complètement parenthétisée, est très peu naturelle, c'est pourquoi nous avons choisi la notation (infixée) du FØRTRAN , estimant qu'elle est "suffisamment lisible".
Il faut toutefois remarquer que les opérateurs "*" , "+" , "/" , "-" , "**" utilisés par le FØRTRAN, ne sont pas des symboles manipulables par le LISP.
Ces constatations montrent que la suite d'opérations à effectuer sur l'expression polynomiale initiale devra être la suivante.

(1) Remplacer les opérateurs FØRTRAN par des symboles atomiques.

(2) Passer de la notation infixée à la notation préfixée.

(3) Dériver l'expression obtenue.

(4) Simplifier le résultat.

(5) Passer de la notation préfixée à la notation infixée.

(6) Remplacer les symboles atomiques représentant des opérateurs par les signes FØRTRAN correspondants.

Nous décrivons, ci-dessous, des fonctions LISP réalisant ces actions.

La fonction "LIRE".

Comme le programme standard de lecture des S-expressions ne reconnaît aucun des symboles "*" , "+" , "-" , "/" , "**" , nous avons écrit en PL/I un sous-programme nommé "LIREFØRM" que nous avons incorporé à l'interpréteur pour constituer la fonction "LIRE" de type SUBR. Cette fonction lit sur le support d'entrée, une expression polynomiale mise entre parenthèses, en remplaçant les opérateurs FØRTRAN par des symboles atomiques conformément au tableau ci-dessous.

FØRTRAN	LISP
*	TIMES
+	PLUS
-	DIFFERENCE
/	QUØTIENT
**	EXPT

Par exemple, l'exécution du programme

(LIRE) ,

suivi sur le support d'entrée de

(3*X + 2) ,

délivre le résultat:

(3,TIMES,X,PLUS,2) .

La fonction "EXPRESSIØN"

traduit en forme préfixée une expression construite d'après la grammaire ci-dessous.

⟨expression⟩::=⟨terme⟩|⟨expression⟩PLUS⟨terme⟩|
⟨expression⟩DIFFERENCE⟨terme⟩.

⟨terme⟩::=⟨secondaire⟩|⟨secondaire⟩TIMES⟨terme⟩|
⟨secondaire⟩QUØTIENT⟨terme⟩.

⟨secondaire⟩::=⟨primaire⟩|⟨primaire⟩EXPT⟨nombre⟩.

⟨primaire⟩::=⟨atome⟩|(⟨expression⟩). (*)

Les fonctions "EXPRESSIØN" , "TERME" , "SECØNDAIRE" se basent sur cette grammaire pour traduire des expressions lues par "LIRE" en forme infixée.
Par exemple, si ε^* est une expression construite d'après la grammaire ci-dessus et mise entre parenthèses,

si ε^* peut s'écrire $(\omega^*,\text{PLUS},\tau^*)$ où ω^* est une expression et τ^* est un terme,

expression$[\varepsilon^*]$ est
list[PLUS;expression$[(\omega^*)]$;terme$[(\tau^*)]]$,

si ε^* peut s'écrire $(\omega^*,\text{DIFFERENCE},\tau^*)$,

expression$[\varepsilon^*]$ est
list[DIFFERENCE;expression$[(\omega^*)]$;terme$[(\tau^*)]]$,

sinon,

expression$[\varepsilon^*]$ est terme$[\varepsilon^*]$.

Il est facile de définir de même les fonctions "TERME" et "SECØNDAIRE" en examinant la grammaire.

Si on présente à l'interpréteur le programme LISP :

(EXPRESSIØN(3,TIMES,X,PLUS,2)) ,

il renverra pour résultat :

(PLUS(TIMES,3,X)2) .

(*) Dans cette grammaire, il n'y a pas d'opérateur "-" monadique.

```
(DEFINE(
(SECONDAIRE (LAMBDA(X)(COND,
   ((NULL(CDR X))(COND((ATOM(CAR X))(CAR X))
             (T (EXPRESSION(CAR X))))))
   (T (LIST(QUOTE EXPT)
            (COND((ATOM(CAR X))(CAR X))
                 (T(EXPRESSION(CAR X))))
            (CADDR X)))
)))
(TERME (LAMBDA(X)(PROG(U,V,W)
         (SETQ U X)
         (SETQ V (CDR U))
   A     (SETQ W (CAR V))
   (COND ((NULL V)(RETURN(SECONDAIRE X)))
         ((TERMP W)(GOQ B)))
         (SETQ U V)
         (SETQ V(CDR V))
         (GOQ A)
   B     (RPLACD U NIL)
         (RETURN (LIST W(SECONDAIRE X)(TERME(CDR V))))
)))
(EXPRESSION(LAMBDA(X)(PROG(U,V,W,Y,Z,YZ)
         (SETQ U X)
         (SETQ V(CDR U))
   A     (SETQ W(CAR V))
   (COND ((NULL V)(RETURN(TERME X)))
         ((EXPRP W)(GOQ B)))
         (SETQ U V)
         (SETQ V (CDR V))
         (GOQ A)
   B     (SETQ Y U)
         (SETQ Z V)
         (SETQ YZ  W)
   C     (SETQ U V)
         (SETQ V(CDR V))
         (SETQ W(CAR V))
   (COND((NULL V)(GOQ D))
         ((EXPRP W)(GOQ B))
         (T(GOQ C)))
   D     (RPLACD Y NIL)
         (RETURN(LIST YZ(EXPRESSION X)(TERME (CDR Z))))
)))
(EXPRP(LAMBDA(X)(OR(EQ X(QUOTE PLUS))(EQ X(QUOTE DIFFERENCE)))))
(TERMP(LAMBDA(X)(OR(EQ X(QUOTE TIMES))(EQ X(QUOTE QUOTIENT)))))
(PLUSP(LAMBDA(X)(OR(EQ X(QUOTE PLUS))(EQ X(QUOTE MINUS)))))
))
  DEBUT DE EVALQUOTE
  FIN DE EVALQUOTE
NIL
  EJECT
```

La fonction "DERIV"

constitue un des exemples les plus connus d'utilisation du LISP, elle est la traduction en LISP, pour des expressions écrites d'après la grammaire de la page V.3 des règles de dérivation ci-dessous.

Etant donnés la constante a , la variable x , le nombre n et les expressions polynomiales y , y_1 et y_2 ,

si $y = a$, $D_x y = 0$,

si $y = x$, $D_x y = 1$,

si $y = y_1 + y_2$, $D_x y = D_x y_1 + D_x y_2$,

si $y = y_1 - y_2$, $D_x y = D_x y_1 - D_x y_2$,

si $y = y_1 * y_2$, $D_x y = y_1 * D_x y_2 + D_x y_1 * y_2$,

si $y = y_1 / y_2$, $D_x y = (D_x y_1 * y_2 - y_1 * D_x y_2) / y_2 ** 2$,

si $y = y_1 ** n$, $D_x y = n * y_1 ** (n-1) * D_x y_1$.

Par exemple, le résultat du programme LISP:

```
(DERIV(PLUS(TIMES,3,X)2))
```

est

```
(PLUS(PLUS(TIMES,X,0)(TIMES,3,1))0) .
```

```
(DEFINE(
(DERIV(LAMBDA(E X)
   (COND((ATOM E)(COND((EQ E X) 1)(T 0)))
   ((EXPRP(CAR E))
        (LIST(CAR E)(DERIV(CADR E)X)(DERIV(CADDR E)X)))
   ((EQ (CAR E)(QUOTE TIMES))
    (LIST (QUOTE PLUS)
     (LIST(CAR E)(CADDR E)(DERIV(CADR E)X))
     (LIST(CAR E)(CADR E)(DERIV(CADDR E)X))))
   ((EQ (CAR E)(QUOTE QUOTIENT))
    (LIST(CAR E)
     (LIST(QUOTE DIFFERENCE)
      (LIST(QUOTE TIMES)(CADDR E)(DERIV(CADR E)X))
      (LIST(QUOTE TIMES)(CADR E)(DERIV(CADDR E)X)))
     (LIST(QUOTE TIMES)(CADDR E)(CADDR E))))
   ((EQ(CAR E)(QUOTE EXPT))
    (LIST(QUOTE TIMES)
     (LIST(QUOTE TIMES)(CADDR E)
   (COND((EQUAL(CADDR E)2)(CADR E))
        (T(LIST(CAR E)(CADR E)(DIFFERENCE(CADDR E)1)))))
    (DERIV(CADR E)X)))
   (T NIL))))
))
 DEBUT DE EVALQUOTE
 FIN DE EVALQUOTE
NIL
 EJECT
```

La fonction "SIMPLIFY".

Cette fonction est tirée de la référence [5]. Nous nous sommes d'ailleurs basés sur cette référence tout au long de cet exemple.
Le but de "SIMPLIFY" est de simplifier le résultat fourni par "DERIV" qui comporte généralement un grand nombre de termes redondants, à tel point que la dérivation automatique non suivie de simplification est d'un intérêt quasi nul. Malheureusement, le problème de la simplification automatique des expressions algébriques est très compliqué; d'ailleurs, la notion de "forme la plus simplifiée" d'une expression algébrique n'est même pas bien définie.

On trouvera dans la référence [5] , les règles de simplification utilisées par "SIMPLIFY". En gros, on peut dire qu'on simplifie l'expression "localement" en ce sens qu'étant donné un noeud de l'arbre équivalent à l'expression écrite en mode préfixé, on simplifie d'abord ses sous-arbres, puis on essaie de donner une forme plus simple à l'ensemble du noeud et des sous-arbres simplifiés, mais on ne compare pas des sous-arbres non adjacents.
Si on considère, par exemple, les deux expressions équivalentes:

(1) (DIFFERENCE(TIMES(TIMES,2,X)1)0) ,

(2)
```
(PLUS(TIMES(PLUS,X,A)(DIFFERENCE,1,0))
     (TIMES(DIFFERENCE,X,A)(PLUS,1,0))) ,
```

la première sera simplifiée de la façon suivante :

(1)

(PLUS(TIMES(TIMES,2,X)1)(MINUS,0))

(PLUS(TIMES,2,X)0)

(TIMES,2,X) ,

et la seconde sera traitée comme suit :

(2)

```
(PLUS(TIMES(PLUS,X,A)(PLUS,1(MINUS,0)))
     (TIMES(PLUS,X(MINUS,A))(PLUS,1,0)))

(PLUS(TIMES(PLUS,X,A)(PLUS,1,0))
     (TIMES(PLUS,X(MINUS,A))1))

(PLUS(TIMES(PLUS,X,A)1)
     (PLUS,X(MINUS,A)))

(PLUS(PLUS,X,A)
     (PLUS,X,(MINUS,A))) .
```

Cette dernière forme ne sera pas simplifiée davantage.

Remarques.

- "SIMPLIFY" remplace partout l'opérateur diadique "DIFFERENCE" par une combinaison des opérateurs "PLUS" et "MINUS" qui est monadique.

- Lorsque "SIMPLIFY" rencontre une expression qui ne contient que des nombres comme opérandes, elle l'exécute. (Par utilisation de la fonction "EVAL".) Par exemple, si elle rencontre

(TIMES,3,5) ,

"SIMPLIFY" fait appel à "EVAL" pour évaluer cette expression, qui est une forme LISP valide, et trouver le résultat "15". C'est l'exemple de programme LISP, créé (éventuellement) puis évalué par un autre programme LISP, annoncé au bas de la page I.20.

```
(DEFINE(
(SIMPLIFY(LAMBDA(E)
   (PROG(A,B,C,D)
   (COND((ATOM E)(RETURN E)))
    (SETQ A(SIMPLIFY(CADR E)))
   (COND((EQ(SETQ C(CAR E))(QUOTE MINUS))
        (RETURN(SMINUS(LIST C A)))))
   (SETQ B (SIMPLIFY(CADDR E)))
   (COND((EQ C(QUOTE DIFFERENCE))
        (RETURN(SPLUS(LIST(QUOTE PLUS)
        (SMINUS(LIST(QUOTE MINUS)B))A)))))
   (SETQ D(LIST C A B))
   (RETURN(COND((EQ C(QUOTE PLUS))(SPLUS D))
                ((EQ C(QUOTE TIMES))(STIMES D))
                ((EQ C(QUOTE QUOTIENT))(SQUOTIENT D))
                ((EQ C (QUOTE EXPT))(SEXPT D))
                (T D)
)))))
(EVENP(LAMBDA(X)(EQUAL X (TIMES(QUOTIENT X 2)2))))
))
  DEBUT DE EVALQUOTE
  FIN DE EVALQUOTE
NIL
  EJECT
```

```
(DEFINE(
((SPLUS(LAMBDA(E)
   (COND((NUMBERP(CADDR E))
         (COND((NUMBERP(CADR E))(EVAL E))
               ((ZEROP(CADDR E))(CADR E))
               ((AND(EQ(CAR(CADR E))(QUOTE MINUS))
                     (NUMBERP(CADR(CADR E))))
               (COND((GREATERP(CADR(CADR E))(CADDR E))
                     (MINUS(DIFFERENCE(CADR(CADR E))(CADDR E))))
                     (T   (DIFFERENCE(CADDR E)(CADR(CADR E))))))
               (T(COLLECT(LIST(CAR E)(CADDR E)(CADR E))))))
         ((ZEROP(CADR E))(CADDR E))
         ((EQUAL(CADR E)(CADDR E))
               (COLLECT(LIST(QUOTE TIMES)2(CADR E))))
         ((AND(NOT(ATOM(CADR E)))(EQ(CAAR(CDR E))(QUOTE MINUS)))
         (COND((AND(NOT(ATOM(CADDR E)))
               (EQ(CAAR(CDDR E))(QUOTE MINUS)))
               (LIST(QUOTE MINUS)
               (COLLECT(LIST(CAR E)
                             (CADR(CADR E))
                             (CADR(CADDR E))))))
               ((EQUAL(CADR(CADR E))(CADDR E))0)
               (T(COLLECT(LIST(CAR E)(CADDR E)(CADR E))))))
         ((AND(NOT(ATOM(CADDR E)))(EQ(CAAR(CDDR E))(QUOTE MINUS)))
         (COND((EQUAL(CADR(CADDR E))(CADR E))0)
               (T(COLLECT E))))
         (T(COLLECT E)))))
(STIMES(LAMBDA(E)
   (COND((NUMBERP(CADDR E))
         (COND((NUMBERP(CADR E))(EVAL E))
               ((ZEROP(CADDR E))0)
               ((EQUAL 1(CADDR E))(CADR E))
               (T(COLLECT(LIST(CAR E)(CADDR E)(CADR E))))))
         ((NUMBERP(CADR E))
         (COND((ZEROP(CADR E))0)
               ((EQUAL 1(CADR E))(CADDR E))
               (T(COLLECT E))))
         ((EQUAL(CADR E)(CADDR E))
          (SEXPT(LIST(QUOTE EXPT)(CADR E)2)))
         ((AND(NOT(ATOM(CADR E)))(EQ(CAR(CADR E))(QUOTE MINUS)))
          (COND((AND(NOT(ATOM(CADDR E)))
               (EQ(CAAR(CDDR E))(QUOTE MINUS)))
         (COLLECT(LIST(CAR E)(CADR(CADR E))(CADR(CADDR E)))))
               ((EQUAL(CADR(CADR E))(CADDR E))
         (LIST(QUOTE MINUS)(LIST(QUOTE EXPT)(CADDR E)2)))
         (T(COLLECT(LIST(CAR E)(CADDR E)(CADR E))))))
         ((AND(NOT(ATOM(CADDR E)))(EQ(CAAR(CDDR E))(QUOTE MINUS)))
         (COND((EQUAL(CADR(CADDR E))(CADR E))
               (LIST(QUOTE MINUS)(LIST(QUOTE EXPT)(CADR E)2)))
               (T(COLLECT E))))
         (T(COLLECT E)))))
))
  DEBUT DE EVALQUOTE
  FIN DE EVALQUOTE
NIL
  EJECT
```

```
(DEFINE(
(COLLECT(LAMBDA(E)
   (COND((ATOM E)E)
         ((ATOM(CADDR E))
         (COND((ATOM(CADR E))E)
               (T(COLLECT(LIST(CAR E)(CADDR E)(CADR E))))))
         ((AND(EQ(CAR E)(CAAR(CDDR E)))(NUMBERP(CADR(CADDR E))))
               (COND((NUMBERP(CADR E))
               (LIST(CAR E)
               (EVAL(LIST(CAR E)(CADR E)(CADR(CADDR E))))
               (CADDR(CADDR E))))
               ((ATOM(CADR E))E)
               ((AND(EQ(CAR E)(CAR(CADR E)))(NUMBERP(CADR(CADR E))))
               (LIST(CAR E)
               (EVAL(LIST(CAR E)(CADR(CADR E))(CADR(CADDR E))))
               (LIST(CAR E)(CADDR(CADR E))
               (CADDR(CADDR E)))))
               (T E))))
         (T E))))
(SQUOTIENT(LAMBDA(E)
   (COND((EQUAL(CADR E)(CADDR E))1)
         ((ZEROP(CADR E))0)
         ((EQUAL 1(CADR E))E)
         ((NUMBERP(CADDR E))
         (COND((NUMBERP(CADR E))(EVAL E))
         ((EQUAL 1(CADDR E))(CADR E))
         (T E)))
         ((AND(NOT(ATOM(CADDR E)))(EQ(CAAR(CDDR E))(QUOTE MINUS)))
         (SMINUS(LIST(QUOTE MINUS)
               (SQUOTIENT(LIST(QUOTE QUOTIENT)(CADR E)(CADR(CADDR E)))))))
         ((AND(NOT(ATOM(CADR E)))(EQ(CAAR(CDR E))(QUOTE MINUS)))
         (LIST(QUOTE MINUS)
               (LIST(QUOTE QUOTIENT)(CADR(CADR E))(CADDR E))))
         (T E))))
(SEXPT(LAMBDA(E)
   (COND((ZEROP(CADDR E))1)
         ((EQUAL 1(CADDR E))(CADR E))
         ((NUMBERP(CADR E))(EVAL E))
         ((ATOM(CADR E))E)
         ((EQ(CAR(CADR E))(QUOTE EXPT))
         (LIST(QUOTE EXPT)(CADR(CADR E))
               (TIMES(CADDR E)(CADDR(CADR E)))))
         ((NOT(EQ(CAR(CADR E))(QUOTE MINUS))) E)
         ((EVENP(CADDR E))
               (SEXPT(LIST(QUOTE EXPT)(CADR(CADR E))(CADDR E))))
         (T(LIST(QUOTE MINUS)
               (SEXPT(LIST(QUOTE EXPT)(CADR(CADR E))(CADDR E)))))))
(SMINUS(LAMBDA(X)(COND,
   ((ZEROP (CADR X))0)
   (T(MINUS(CADR X)))
)))
))
 DEBUT DE EVALQUOTE
 FIN DE EVALQUOTE
NIL
 EJECT
```

La fonction "INFIXER"
---

transforme une expression écrite en notation préfixée, en une expression écrite en notation infixée, d'après une syntaxe identique à celle de la page V.5, sauf que l'opérateur diadique "DIFFERENCE" est remplacé par l'opérateur "MINUS" qui est monadique ou diadique suivant les cas. Cette fonction transforme les expressions en essayant d'obtenir un résultat contenant le moins de parenthèses possible. Toutefois, certains cas mal conditionnés seront transformés avec des parenthèses inutiles car on a voulu écrire un algorithme relativement simple.

Remarques.

- "INFIXER" utilise la pseudo-fonction "CØNC" qui concatène un nombre quelconque de listes, ce qui donne un exemple d'utilisation des fonctions de type FEXPR. "CØNC", elle-même, se sert de la fonctionnelle "MAPCØN" ce qui donne un exemple d'utilisation des arguments fonctionnels. "NCØNC" est une fonction qui concatène deux listes.

- La fonction "CØNC" est définie au moyen de la pseudo-fonction "DEFLIST" qui joue le même rôle que "DEFINE" à part qu'elle peut mettre sur la p-liste des atomes qu'elle définit, n'importe quel indicateur. (Mais les seuls cas ayant une signification sont "EXPR" , " APVAL" , "FEXPR".)

Exemple :

Le traitement du programme LISP

```
(INFIXER(PLUS(PLUS,X,A)
              (PLUS,X(MINUS,A)))) ,
```

donnera le résultat suivant:

```
(X,PLUS,A,PLUS,X,MINUS,A) .
```

```
(DEFINE(
(MAPCON(LAMBDA(X,FN)(COND,
        ((NULL X)NIL)
        (T (NCONC(FN X)(MAPCON(CDR X)(FUNCTION FN))))
)))
(NCONC(LAMBDA(X,Y)(PROG(U)
        (SETQ U X)
    A   (COND ((NULL(CDR U))(RPLACD U Y))(T(GOQ B)))
        (RETURN X)
    B   (SETQ U (CDR U))
        (GOQ A)
)))
(DEFLIST(LAMBDA(X Y)(PROG(U),
    A   (COND ((NULL X)(RETURN X)))
        (SETQ U (CAR X))
        (RPLACD (CAR U) (CONS Y(CADR U)))
        (SETQ X (CDR X))
        (GOQ A)
)))
))
 DEBUT DE EVALQUOTE
 FIN DE EVALQUOTE
NIL

(DEFLIST(
(CONC(LAMBDA(X,Y)
        (MAPCON X(FUNCTION (LAMBDA(X)(EVAL(CAR X) Y))))
))
)FEXPR
)
 DEBUT DE EVALQUOTE
 FIN DE EVALQUOTE
NIL
 EJECT
```

```
(DEFINE(
((INFIXER(LAMBDA(X)(PROG(U)
   (RETURN,
   (COND,
        ((ATOM X)(LIST X))
        ((EQ(SETQ U(CAR X))(QUOTE MINUS))(IMINUS X))
        ((EQ U(QUOTE PLUS))(IPLUS X))
        ((TERMP U)(ITERM X))
        (T (ISECO X))))
)))
((IMINUS(LAMBDA(X)(COND,
        ((ATOM(CADR X))X)
        (T (LIST(QUOTE MINUS)(INFIXER(CADR X))))
)))
((ISECO(LAMBDA(X)(LIST,
   (COND((ATOM(CADR X))(CADR X))
        (T(INFIXER(CADR X))))
   (QUOTE EXPT)
   (CADDR X))))
(ITERM(LAMBDA(X)(CONC,
   (COND((PLUSP(CAR(CADR X)))(LIST(INFIXER(CADR X))))
        (T(INFIXER(CADR X))))
   (LIST(CAR X))
   (COND((PLUSP(CAR(CADDR X)))(LIST(INFIXER(CADDR X))))
        (T (INFIXER(CADDR X))))
)))
((IPLUS(LAMBDA(X)(PROG(U,V)
   (SETQ U ,
   (COND((EQ(CAR(CADDR X))(QUOTE MINUS))(QUOTE MINUS))
        (T (QUOTE PLUS))))
   (COND((EQ U(QUOTE MINUS))
        (COND,
             ((OR(MEMBER(QUOTE PLUS)(SETQ V(INFIXER(CADR(CADDR X)))))
                       (MEMBER(QUOTE MINUS)V))
             (SETQ V (LIST V)))
             (T NIL)))
        (T(SETQ  V(INFIXER(CADDR X))))))
   (RETURN(CONC(INFIXER(CADR X))(LIST U)V))
)))
))
  DEBUT DE EVALQUOTE
  FIN DE EVALQUOTE
NIL
  EJECT
```

La fonction "ECRIRE".

Cette fonction permet d'imprimer, dans une notation semblable à celle du FØRTRAN, la liste fournie par "INFIXER". Elle est du type SUBR pour la même raison que "LIRE"; elle correspond au sous-programme "PRINTFØRM" qui est beaucoup plus simple que "PRINTEXPR" parce qu'il ne traite que des listes.
La table de correspondance opérateur FØRTRAN / symbole atomique, est identique à celle utilisée par "LIRE" sauf que le symbole "DIFFERENCE" est remplacé par "MINUS" .

La fonction "DERIVØNS"
---

dérive une expression polynomiale écrite dans la notation du FØRTRAN. Comme le programme standard de lecture ne peut pas lire de telles expressions, le problème à traiter doit être présenté à l'interpréteur autrement que sous la forme habituelle : il faut lui fournir les trois expressions ci-dessous.

(DERIVØNS)	fonction sans argument.
(ε*)	expression à dériver.
(x*)	variable par rapport à laquelle la dérivation s'effectue.

La fonction "DERIVØNS" lit l'expression polynomiale et la variable, effectue la suite d'opérations décrite précédemment et imprime le résultat et quelques commentaires. Finalement, on obtient le résultat suivant sur le listing de sortie.

```
(DERIVØNS)
DEBUT DE EVALQUØTE
( LA DERIVEE DE )
( ε* )
( PAR RAPPØRT A )
( x* )
( EST )
( ε'* )
FIN DE EVALQUØTE
NIL
```

```
(DEFINE(
(DERIVONS(LAMBDA NIL(PROG(U,V)
   (PRINT(QUOTE(LA DERIVEE DE)))
   (SETQ U(LIRE))
   (PRINT(QUOTE(PAR RAPPORT A)))
   (SETQ V(CAR(LIRE)))
   (PRINT(QUOTE(EST)))
   (ECRIRE(INFIXER(SIMPLIFY(DERIV(EXPRESSION U)V))))
   (RETURN NIL)
))) ))
 DEBUT DE EVALQUOTE
 FIN DE EVALQUOTE
NIL
 EJECT
```

```
  (DERIVONS)
  DEBUT DE EVALQUOTE
( LA DERIVEE  DE )
( X**4 - 10*X**3 + 35*X**2 - 17*X + 24 )
( PAR  RAPPORT  A  )
( X )
( EST  )
( 4 * X  **3 -  30 * X  **2 +  70 * X  -  17 )
  FIN DE EVALQUOTE
NIL
  RECLAIM,LISP
  GARBAGE COLLECTOR

  (DERIVONS)
  DEBUT DE EVALQUOTE
( LA DERIVEE  DE )
( (X**4 + 3*X**2 + 4)/(1 + X**2) )
( PAR  RAPPORT  A  )
( X )
( EST  )
( ( ( 1 + X  **2 ) * ( 4 * X  **3 + 6 * X  ) - ( 4 + X  **4 + 3 * X  **2 ) * 2 * X  ) / ( 1 + X  **2 ) **2 )
  FIN DE EVALQUOTE
NIL
  RECLAIM,LISP
  GARBAGE COLLECTOR

  (DERIVONS)
  DEBUT DE EVALQUOTE
( LA DERIVEE  DE )
(X**2 + 2 + 2/(1 + X**2) )
( PAR  RAPPORT  A  )
( X )
( EST  )
( 2 * X  - 4 * X  / ( 1 + X  **2 ) **2 )
  FIN DE EVALQUOTE
NIL
  EJECT
```

La fonction "DIFF"
---

effectue les mêmes opérations que "DERIVØNS", mais imprime aussi tous les résultats intermédiaires.

La fonction "DERIV2"
---

calcule la dérivée seconde d'une expression, par rapport à deux variables à spécifier, distinctes ou non. Son exécution est très lente car on ne simplifie pas l'expression obtenue avant de dériver pour la seconde fois. (Parce que "DERIV" ne reconnaît pas l'atome "MINUS".)

```
(DEFINE(
(DIFF(LAMBDA NIL (PROG(U,V)
   (PRINT(QUOTE(LA DERIVEE DE)))
   (SETQ U(LIRE))
   (PRINT(QUOTE(PAR RAPPORT A)))
   (SETQ V(CAR(LIRE)))
   (PRINT U)
   (SETQ U(PRINT(EXPRESSION U)))
   (SETQ U(PRINT(DERIV U V)))
   (SETQ U(PRINT(SIMPLIFY U)))
   (SETQ U(PRINT(INFIXER U)))
   (PRINT(QUOTE(EST)))
   (ECRIRE U)
   (RETURN NIL)
)))
))
 DEBUT DE EVALQUOTE
 FIN DE EVALQUOTE
NIL
 EJECT
```

```
  (DIFF)
  DEBUT DE EVALQUOTE
( LA DERIVEE  DE )
( (A - (X - B - (X- C - (X + D)))) )
( PAR  RAPPORT  A  )
( X )
( ( A  DIFFERENCE ( X  DIFFERENCE B  DIFFERENCE ( X  DIFFERENCE C  DIFFERENCE ( X  PLUS D  ) ) ) ) )
( DIFFERENCE A  ( DIFFERENCE ( DIFFERENCE X  B  ) ( DIFFERENCE ( DIFFERENCE X  C  ) ( PLUS X  D  ) ) ) )
( DIFFERENCE 0 ( DIFFERENCE ( DIFFERENCE 1 0 ) ( DIFFERENCE ( DIFFERENCE 1 0 ) ( PLUS 1 0 ) ) ) )
( MINUS  1 )
( MINUS  1 )
( EST  )
( - 1 )
  FIN DE EVALQUOTE
NIL
  RECLAIM,LISP
  GARBAGE COLLECTOR

  (DIFF)
  DEBUT DE EVALQUOTE
( LA DERIVEE  DE )
( 3**3*(A + B)**3 + 2**2*(A + B)**2 - (A + B) )
( PAR  RAPPORT  A  )
( A )
( 3 EXPT 3 TIMES  ( A  PLUS B  ) EXPT 3 PLUS 2 EXPT 2 TIMES  ( A  PLUS B  ) EXPT 2 DIFFERENCE ( A  PLUS B  ) )
( DIFFERENCE ( PLUS ( TIMES  ( EXPT 3 3 ) ( EXPT ( PLUS A  B  ) 3 ) ) ( TIMES  ( EXPT 2 2 ) ( EXPT ( PLUS A
B  ) 2 ) ) ) ( PLUS A  B  ) )
( DIFFERENCE ( PLUS ( PLUS ( TIMES  ( EXPT ( PLUS A  B  ) 3 ) ( TIMES  ( TIMES  3 ( EXPT 3 2 ) ) 0 ) ) ( TIMES  (
EXPT 3 3 ) ( TIMES  ( TIMES  3 ( EXPT ( PLUS A  B  ) 2 ) ) ( PLUS 1 0 ) ) ) ) ( PLUS ( TIMES  ( EXPT ( PLUS A
B  ) 2 ) ( TIMES  ( TIMES  2 2 ) 0 ) ) ( TIMES  ( EXPT 2 2 ) ( TIMES  ( TIMES  2 ( PLUS A  B  ) ) ( PLUS 1 0 ) ) ) ) )
( PLUS 1 0 ) )
( PLUS ( PLUS ( TIMES  81 ( EXPT ( PLUS A  B ) 2 ) ) ( TIMES  8 ( PLUS A  B  ) ) ) ( MINUS  1 ) )
(  81 TIMES  ( A  PLUS B  ) EXPT 2 PLUS 8 TIMES  ( A  PLUS B  ) MINUS  1 )
( EST  )
(  81 * ( A  + B  ) **2 + 8 * ( A  + B  ) - 1 )
  FIN DE EVALQUOTE
NIL
  RECLAIM,LISP
  GARBAGE COLLECTOR
  EJECT
```

```
(DEFINE(
(DERIV2(LAMBDA NIL (PROG(U,V,W)
   (PRINT(QUOTE(LA DERIVEE SECONDE DE)))
   (SETQ U(LIRE))
   (PRINT(QUOTE(PAR RAPPORT AUX DEUX VARIABLES)))
   (SETQ V(CAR(SETQ W(LIRE))))
   (SETQ W(CADR W))
   (PRINT(QUOTE( N EST AUTRE QUE)))
   (ECRIRE(INFIXER(SIMPLIFY(DERIV(DERIV(EXPRESSION U)V)W))))
   (RETURN(QUOTE OK))
))) ))
  DEBUT DE EVALQUOTE
  FIN DE EVALQUOTE
NIL
  RECLAIM,LISP
  GARBAGE COLLECTOR
  EJECT
```

```
  (DERIV2)
  DEBUT DE EVALQUOTE
( LA DERIVEE  SECONDE  DE )
( 1 + X - Y + X**2 - 2*X*Y + Y**2 + X**3 - 3*X**2*Y + 3*X*Y**2 - Y**3 )
( PAR  RAPPORT  AUX  DEUX VARIABLES  )
( X,X )
( N  EST  AUTRE  QUE  )
  GARBAGE COLLECTOR
  GARBAGE COLLECTOR
( 2 + 6 * X  - 6 * Y  )
  FIN DE EVALQUOTE
OK
  RECLAIM,LISP
  GARBAGE COLLECTOR

  (DERIV2)
  DEBUT DE EVALQUOTE
( LA DERIVEE  SECONDE  DE )
( 1 + X - Y + X**2 - 2*X*Y + Y**2 + X**3 - 3*X**2*Y + 3*X*Y**2 - Y**3 )
( PAR  RAPPORT  AUX  DEUX VARIABLES  )
( X,Y )
( N  EST  AUTRE  QUE  )
  GARBAGE COLLECTOR
  GARBAGE COLLECTOR
( 6 * Y  - ( 2 + 6 * X  ) )
  FIN DE EVALQUOTE
OK
  RECLAIM,LISP
  GARBAGE COLLECTOR

  (DERIV2)
  DEBUT DE EVALQUOTE
( LA DERIVEE  SECONDE  DE )
( 1 + X - Y + X**2 - 2*X*Y + Y**2 + X**3 - 3*X**2*Y + 3*X*Y**2 - Y**3 )
( PAR  RAPPORT  AUX  DEUX VARIABLES  )
( Y,X )
( N  EST  AUTRE  QUE  )
  GARBAGE COLLECTOR
  GARBAGE COLLECTOR
( 6 * Y  - ( 2 + 6 * X  ) )
  FIN DE EVALQUOTE
OK
  RECLAIM,LISP
  GARBAGE COLLECTOR

  (DERIV2)
  DEBUT DE EVALQUOTE
( LA DERIVEE  SECONDE  DE )
( 1 + X - Y + X**2 - 2*X*Y + Y**2 + X**3 - 3*X**2*Y + 3*X*Y**2 - Y**3 )
( PAR  RAPPORT  AUX  DEUX VARIABLES  )
( Y,Y )
( N  EST  AUTRE  QUE  )
  GARBAGE COLLECTOR
  GARBAGE COLLECTOR
( 2 + 6 * X  - 6 * Y  )
  FIN DE EVALQUOTE
OK
  EJECT
```

APPENDICE : LISTING DE L'INTERPRETEUR.

On trouvera dans les pages qui suivent le listing complet de notre interpréteur écrit en PL/I.

```
SPPL1..PROCEDURE OPTIONS(MAIN),.

      LISPPL1..PROCEDURE OPTIONS(MAIN),.
      DCL  INPUT FILE RECORD SEQUENTIAL INPUT BUFFERED
           ENV(CONSECUTIVE),.
      DCL  PRINT FILE RECORD SEQUENTIAL OUTPUT BUFFERED
           ENV(CONSECUTIVE),.
      DCL  ALINK(30000)   BIN FIXED(15) STATIC INIT(
      201,203,205,207,209,210,213,347,350,218,
      220,222,224,226,228,230,232,234,237,240,243,246,249,253,257,
      258,260,262,264,266,268,270,273,276,279,282,285,288,292,296,301,
      353,356,358,360,(5)80,
      215,216,340,342,312,321,344,333,316,
      (10)80,
      318,330,327,
      (7)80,
      310,309,308,314,337,324,305,
      (4)80,
      101,111,126,151,171,186,(4)80,
      5,51,25,81,82,
      (5)80,
      2,3,10,26,27,52,53,80,
      (7)80,
      1,4,7,11,12,13,14,15,16,17,28,29,30,31,54,55,59,83,
      43,44,(5)80,
      18,19,32,33,34,35,36,84,85,56,57,70,
      45,(7)80,
      6,8,9,20,21,22,37,71,72,86,
      42,(4)80,
      23,24,38,39,40,41,58
      ),.
      DCL  BLINK(30000)   BIN FIXED(15) STATIC INIT(
      (89)80,
      91,92,93,94,95,96,97,98,99,100,80,
      102,103,104,105,
      (6)80,
      112,113,114,115,116,117,118,
      (8)80,
      127,128,129,130,131,132,133,134,135,136,137,138,139,140,141,142,143,
      144,145,(6)80,
      152,153,154,155,156,157,158,159,160,161,162,
      163,(8)80,
      172,173,174,175,176,177,178,179,180,
      181,(5)80,
      187,188,189,190,191,192,
      (9)80,
      202,80,204,80,206,80,208,80,80,211,212,80,214,80,80,
      217,80,219,80,221,80,223,80,225,80,227,80,229,80,
      231,80,233,80,235,236,80,238,239,80,241,242,80,244,245,
      80,247,248,80,250,251,252,80,254,255,256,80,80,259,
      80,261,80,263,80,265,80,267,80,269,80,271,272,80,274,275,80,277,278,
      80,280,281,80,283,284,80,286,287,80,289,290,291,80,
      293,294,295,80,297,298,299,300,80,302,303,304,80,
      306,307,80,80,80,311,80,313,80,315,80,317,80,319,320,
      80,322,323,80,325,326,80,328,329,80,331,332,80,334,335,336,80,
      338,339,80,341,80,343,80,345,346,80,348,349,80,351,352,80 ,
      354,355,80,357,80,359,80,361,362,80
      ),.
      DCL  INFO (30000)   CHAR(2) BASED(POINTER),.
      DCL  STRING CHAR(400) BASED(POINTER),.
      DCL  BUF3           CHAR(30) STATIC,.
      DCL  (FREE,MINFREE) BIN FIXED(15) STATIC INIT(401),.
      DCL MAXFREE         BIN FIXED(15) STATIC INIT(30000),.
      DCL  PSTACK(50)     BIN FIXED(15) STATIC,.
```

```
%PPL1..PROCEDURE OPTIONS(MAIN),.

      DCL  UN              BIN FIXED(15) STATIC INIT(1),.
      DCL  DEUX            BIN FIXED(15) STATIC INIT(2),.
       DCL  LABELL         BIN FIXED(15) STATIC INIT(70),.
      DCL  LAMBDA          BIN FIXED(15) STATIC INIT(71),.
      DCL  FUNARG          BIN FIXED(15) STATIC INIT(72),.
      DCL  NIL             BIN FIXED(15) STATIC INIT(80),.
      DCL  T               BIN FIXED(15) STATIC INIT(81),.
      DCL  F               BIN FIXED(15) STATIC INIT(82),.
      DCL  EXPR            BIN FIXED(15) STATIC INIT(83),.
      DCL  FEXPR           BIN FIXED(15) STATIC INIT(84),.
      DCL  APVAL           BIN FIXED(15) STATIC INIT(85),.
      DCL  NUMBER          BIN FIXED(15) STATIC INIT(86),.
      DCL  GO              BIN FIXED(15) STATIC INIT(25),.
      DCL  SET             BIN FIXED(15) STATIC INIT(27),.
      DCL  CSET            BIN FIXED(15) STATIC INIT(7),.
      DCL  QUOTE           BIN FIXED(15) STATIC INIT(56),.
      DCL  RETOUR3         BIN FIXED(15) STATIC INIT(3),.
      DCL  RETOUR4         BIN FIXED(15) STATIC INIT(4),.
      DCL  RETOUR5         BIN FIXED(15) STATIC INIT(5),.
      DCL  RETOUR6         BIN FIXED(15) STATIC INIT(6),.
      DCL  PPO             BIN FIXED(15) STATIC INIT(90),.
      DCL  MAXLINE         BIN FIXED(15) STATIC INIT(60),.
      DCL  LISTN           BIN FIXED(15) STATIC INIT(80),.
      DCL  NUL             BIN FIXED(15) STATIC INIT(32001),.
      DCL  PLUS            BIN FIXED(15) STATIC INIT(30),.
      DCL  TIMES           BIN FIXED(15) STATIC INIT(32),.
      DCL  QUOTIENT        BIN FIXED(15) STATIC INIT(38),.
      DCL  DIFFERENCE      BIN FIXED(15) STATIC INIT(40),.
      DCL  EXPT            BIN FIXED(15) STATIC INIT(44),.
      DCL  MINUS           BIN FIXED(15) STATIC INIT(45),.
      DCL  LISTI(6)       BIN FIXED(15) STATIC INIT(101,111,126,151,171,186),.
      DCL  LISTJ(6)       BIN FIXED(15) STATIC INIT(105,118,143,162,180,192),.
      DCL  BUF1 CHAR(80) STATIC,.
      DCL  BUF2 CHAR(130) STATIC,.
      DCL  1 BOF2 DEFINED BUF2,
           2 CTL BIT(8),
           2 REST CHAR(129),.
           CTL='00001001'B,.
      DCL  SAUT CHAR(8) STATIC,.
      DCL  1 SOT DEFINED SAUT,
            2 SKIP BIT(8),
            2 REST CHAR(7),.
            SKIP='10001011'B,.
      DCL  BUF(16) CHAR(2) STATIC,.
            DCL  BOF(16)  BIN FIXED(15) BASED(POINTEUR),.
            POINTEUR=ADDR(BUF),.
      DCL  (P,Q,R,ACCU,EPSI,STACK,STAET,LIST,ALIST,PHI,PP,PQ,PR,
            PS,PTT,PZ,PZZ,PPQ,QQ,SO,PO,PI,PK,PJ,LINE,I,J,K,L,M,N,
            INDIC,ETAT,RS,GOLIST)
                        BIN FIXED(15) STATIC,.
      DCL  S CHAR(1) STATIC,.
            POINTER=ADDR(ALINK(201)),.
            STRING='ATOMCAR CDR CONSEQDEFINECSETORAND NOT CAARCADRCDARCOPYCDD
RLISTNULLCADDR EQUAL RPLACARPLACDMEMBERRECLAIM REVERSE GODUP SET ADD1SU
B1PLUSEVALTIMES LESSP ZEROP PRINT APPLY RETURNQUOTIENTNUMBERP DIFFERENC
EGREATERPNUMBERF T NIL CONDEXPRPROGLABEL QUOTE APVAL FUNARGLAMBDAFUNCTI
ONFEXPR GOQ SETQCSETQ APPENDLENGTHECRIRELIREEXPTMINUS ',.
            POINTER=ADDR(ALINK),.
            OPEN FILE(INPUT),.OPEN FILE(PRINT),.
            /* SUPERVISEUR */
      EJECT..LINE=MAXLINE,.
      DEBUTLISP..CALL READ,.
```

```
SPPL1..PROCEDURE OPTIONS(MAIN),.

          IF BUF1='  'THEN DO,.
          CALL LIREXPR,.
          CALL EVALQUOTE,.PO=ACCU,.
          ACCU=NIL,.CALL PRINTEXPR,.END,.
          ELSE DO,.
          IF BUF1='  EJECT' THEN GO TO EJECT,.
          ELSE DO,.
          IF BUF1='  RECLAIM,LISP' THEN DO,.
          DO I=97 TO 100,.ALINK(I)=NIL,.END,.
          CALL GARBCOL,.END,.
          ELSE DO,.
          IF BUF1='  FIN,LISP' THEN GOTO FINLISP,.
          ELSE DO,.
          IF BUF1='  DEBUT,LISP' THEN ,.
          ELSE DO,.
          IF BUF1='  REINIT,LISP'THEN DO,.
          DO I=1 TO 6,.J=LISTJ(I),.BLINK(J)=NIL,.END,.
          LISTN=NIL,.FREE=MINFREE,.END,.
          ELSE DO,.
          CALL PRINTCOMMENT('   ERREUR,CARTE CONTROLE       '),.
          GOTO DEBUTLISP,.END,.END,.
          DO I=MINFREE TO MAXFREE,.BLINK(I)=I+1,.END,.
          BLINK(MAXFREE)=NIL,.
          END,.END,.END,.END,.GOTO DEBUTLISP,.
     EXFREE..PROCEDURE(X,Y,Z),.
          DCL (X,Y,Z) BIN FIXED(15),.
          DCL SWITCH(4) LABEL,.
     A..  IF FREE NE NIL THEN DO,.
          Q=FREE,.FREE=BLINK(Q),.
          ALINK(Q)=X,.BLINK(Q)=Y,.Z=Q,.RETURN,.END,.
          GOTO SWITCH(INDIC),.
     SWITCH(1)..
          ALINK(97)=QO,.ALINK(98)=SO,.ALINK(99)=STACK,.ALINK(100)=ACCU,.
          GOTO B,.
     SWITCH(2)..
          ALINK(97)=PO,.ALINK(98)=STAET,.ALINK(99)=STACK,.ALINK(100)=ACCU,.
          GOTO B,.
     SWITCH(3)..
          ALINK(97)=PHI,.ALINK(98)=ALIST,.ALINK(99)=STACK,.ALINK(100)=LIST,.
          GOTO B,.
     SWITCH(4)..
          ALINK(97)=EPSI,.ALINK(98)=ALIST,.ALINK(99)=STACK,.ALINK(100)=ACCU,.
     B..  CALL GARBCOL,.GOTO A,.
          END EXFREE,.
     POP1..PROCEDURE(X,Y),.
          DCL (X,Y) BIN FIXED(15),.
          X=ALINK(Y),.Y=BLINK(Y),.
          END POP1,.
          /* GARBAGE COLLECTOR */
     MARKATOM..PROCEDURE,.
     A..  PPQ=BLINK(Q),.
          IF PPQ NE NIL THEN DO ,.
          BLINK(Q)=-PPQ,.Q=PPQ,.GOTO A,.END,.
          BLINK(Q)=-PPQ,.
          END MARKATOM,.
     GARBCOL..PROCEDURE,.
          DCL TEST BIN FIXED,.
          TEST=1,.
          CALL PRINTCOMMENT('   GARBAGE COLLECTOR          '),.
          PS=0,.PR=PPO,.PTT=NUL,.
     A..  PZ=BLINK(PR),.PZZ=ALINK(PR),.
          IF PZ LT 0 THEN,.
```

```
PPL1..PROCEDURE OPTIONS(MAIN),.

          ELSE DO,.
          IF PZZ LT 0 THEN DO,.Q=PR,.CALL MARKATOM,.END,.
          ELSE DO,.
          BLINK(PR)=-PZ,.
          IF PS LT 50 THEN DO,.
          PS=PS+1,.PSTACK(PS)=PZ,.PR=PZZ,.GOTO A,.END,.
          ELSE DO,.
      C.. PQ=ALINK(PR),.PZ=BLINK(PQ),.
          IF PZ LT 0 THEN,.
          ELSE DO,.
          IF ALINK(PQ) LT 0 THEN DO,.
          Q=PQ,.CALL MARKATOM,.END,.
          ELSE DO,.
          BLINK(PQ)=-PZ,.ALINK(PR)=-PTT,.PTT=PR,.PR=PQ,.GOTO C,.
          END,.END,.
      D.. PQ=-BLINK(PR),.PZ=BLINK(PQ),.
          IF PZ LT 0 THEN,.
          ELSE DO,.
          BLINK(PQ)=-PZ,.BLINK(PR)=-PTT,.PTT=PR,.PR=PQ,.GOTO C,.
          END,.
      E.. IF PTT=NUL THEN,.
          ELSE DO,.
          PQ=PTT,.PZ=ALINK(PQ),.
          IF PZ LT 0 THEN DO,.
          PTT=-PZ,.ALINK(PQ)=PR,.PR=PQ,.GOTO D,.END,.
          PTT=-BLINK(PQ),.BLINK(PQ)=-PR,.PR=PQ,.GOTO E,.END,.
          END,.END,.END,.
          IF PS GT 0 THEN DO,.
          PR=PSTACK(PS),.PS=PS-1,.GOTO A,.END,.
          DO PI=MAXFREE TO MINFREE BY -1,.
          PK=BLINK(PI),.
          IF PK LT 0 THEN BLINK(PI)=-PK,.
          ELSE DO,.
          TEST=0,.
          PK=PI,.BLINK(PI)=NIL,.GOTO H,.END,.END,.
      H.. DO PI=MINFREE TO MAXFREE,.
          PQ=BLINK(PI),.
          IF PQ LT 0 THEN BLINK(PI)=-PQ,.
          ELSE DO,.
          FREE=PI,.GOTO O,.END,.END,.
      O.. PJ=FREE,.
          DO PI=FREE TO PK,.
          PS=BLINK(PI),.
          IF PS LT 0 THEN BLINK(PI)=-PS,.
          ELSE DO,.
          BLINK(PJ)=PI,.PJ=PI,.END,.END,.
          DO PI=1 TO MINFREE,.
          PS=BLINK(PI),.
          IF PS LT 0 THEN BLINK(PI)=-PS,.END,.
          IF TEST NE 0 THEN DO,.
          CALL PRINTCOMMENT('   MEMOIRE INSUFFISANTE        '),.
          STOP,.END,.
          END GARBCOL,.
          /* ROUTINE DE SORTIE */
 PRINTFORM..PROCEDURE,.
          INDIC,K=2,.STACK,STAET=NIL,.R=PO,.
          CALL PRINTCHAR2('( '),.
      A.. P=ALINK(R),.Q=ALINK(P),.PP=ALINK(Q),.R=BLINK(R),.
          IF PP LT 0 THEN DO,.
          IF P=PLUS THEN CALL PRINTCHAR2('+ '),.
          ELSE DO,.
          IF P=MINUS THEN CALL PRINTCHAR2('- '),.
```

```
SPPL1..PROCEDURE OPTIONS(MAIN),.

        ELSE DO,.
        IF P=QUOTIENT THEN CALL PRINTCHAR2('/ '),.
        ELSE DO,.
        IF P=TIMES THEN CALL PRINTCHAR2('* '),.
        ELSE DO,.
        IF P=EXPT THEN CALL PRINTCHAR2('**'),.
        ELSE DO,.
        IF BLINK(P)=NUMBER THEN CALL PRINTNBR,.ELSE CALL PRINTATOM,.
        END,.END,.END,.END,.END,.
B..     IF R=NIL THEN DO,.
        CALL PRINTCHAR2(') '),.
        IF STAET=NIL THEN DO,.CALL EDIT,.RETURN,.END,.
        CALL POP1(R,STAET),.GOTO B,.END,.END,.
        ELSE DO,.
        CALL PRINTCHAR2('( '),.
        CALL EXFREE(R,STAET,STAET),.R=P,.END,.
        GOTO A,.
        END PRINTFORM,.
PRINTEXPR..PROCEDURE,.
        DCL SWATCH(4) LABEL,.
        INDIC,K=2,.STACK,STAET=NIL,.P=PO,.
A..     Q=ALINK(P),.PP=ALINK(Q),.
        IF PP LT 0 THEN DO,.
        IF BLINK(P)=NUMBER THEN CALL PRINTNBR,.ELSE CALL PRINTATOM,.
        IF STAET=NIL THEN GOTO C,.
        IF STACK NE NIL THEN CALL POP1(P,STACK),.END,.
        ELSE DO,.
        CALL PRINTCHAR2('( '),.Q=P,.
AA..    Q=BLINK(Q),.
        IF Q=NIL THEN CALL EXFREE(DEUX,STAET,STAET),.
        ELSE DO,.
        PP=ALINK(Q),.
        IF ALINK(PP) LT 0 THEN CALL EXFREE(UN,STAET,STAET),.
        ELSE GOTO AA,.END,.END,.
B..     ETAT=ALINK(STAET),.GOTO SWATCH(ETAT),.
SWATCH(1)..
        CALL EXFREE(BLINK(P),STACK,STACK),.
        P=ALINK(P),.ALINK(STAET)=3,.GOTO A,.
SWATCH(2)..
        IF P=NIL THEN DO,.
        CALL PRINTCHAR2(') '),.
        IF STACK=NIL THEN GOTO C,.
        STAET=BLINK(STAET),.CALL POP1(P,STACK),.
        GOTO B,.END,.
        CALL EXFREE(BLINK(P),STACK,STACK),.
        P=ALINK(P),.GOTO A,.
SWATCH(3)..
        CALL PRINTCHAR2('. '),.ALINK(STAET)=4,.GOTO A,.
SWATCH(4)..
        CALL PRINTCHAR2(') '),.STAET=BLINK(STAET),.
        IF STAET NE NIL THEN GOTO B,.
C..     CALL EDIT,.
        END PRINTEXPR,.
PRINTCHAR2..PROCEDURE(CHAR2),.
        DCL CHAR2 CHAR(2),.
        IF K GE 120 THEN DO,.CALL EDIT,.K=2,.END,.
        SUBSTR(BUF2,K,2)=CHAR2,.K=K+2,.
       .END PRINTCHAR2,.
PRINTATOM..PROCEDURE,.
        IF K GT 110 THEN DO,.CALL EDIT,.K=2,.END,.
A..     IF Q NE NIL THEN DO,.
        SUBSTR(BUF2,K,2)=INFO(Q),.Q=BLINK(Q),. K=K+2,.GOTO A,.END,.
```

```
SPPL1..PROCEDURE OPTIONS(MAIN),.

        K=K+1,.
        END PRINTATOM,.
    PRINTNBR..PROCEDURE,.
        DCL IJ PIC'ZZZZ9',.
        DCL GNB CHAR(5),.
        I=-PP-1,.IJ=I,.GNB=IJ,.
        IF K GT 110 THEN DO,.CALL EDIT,.K=2,.END,.
        IF I LT 10 THEN DO,.
        SUBSTR(BUF2,K,1)=SUBSTR(GNB,5,1),.K=K+2,.END,.
        ELSE DO,.
        IF I GT 999 THEN DO,.
        SUBSTR(BUF2,K,5)=GNB,.K=K+6,.END,.
        ELSE DO,.
        SUBSTR(BUF2,K,3)=SUBSTR(GNB,3,3),.K=K+4,.END,.
        END,.
        END PRINTNBR,.
    EDIT..PROCEDURE,.
        IF LINE GE MAXLINE THEN DO,.
        WRITE FILE(PRINT) FROM(SAUT),.LINE=1,.END,.
        WRITE FILE(PRINT)FROM(BUF2),.LINE=LINE+1,.
        BOF2.REST=' ',.
        END EDIT,.
    PRINTCOMMENT..PROCEDURE(X),.
        DCL X CHAR(30),.
        BUF3=X,.
        IF LINE GE MAXLINE THEN DO,.
        WRITE FILE(PRINT) FROM(SAUT),.LINE=1,.END,.
        WRITE FILE(PRINT) FROM(BUF3),.LINE=LINE+1,.
        END PRINTCOMMENT,.
        /* ROUTINE DE LECTURE */
     LIRFORM..PROCEDURE,.
        K=80,.STACK=NIL,.INDIC=1,.
    A..  CALL LIRSYMB,.
        IF S='(' THEN DO,.
        CALL EXFREE(NIL,NIL,QO),.P=QO,.
        CALL EXFREE(NIL,STACK,STACK),.
        ETAT=0,.END,.
        ELSE DO,.K=K+1,.GOTO A,.END,.
    B..  K=K+1,.CALL LIRSYMB,.
        IF S='(' THEN DO,.
        IF ETAT=0 THEN DO,.
        CALL EXFREE(NIL,NIL,R),.ALINK(P)=R,.
        CALL EXFREE(P,STACK,STACK),.P=R,.END,.
        ELSE DO,.
        CALL EXFREE(NIL,NIL,R),.BLINK(P)=R,.
        CALL EXFREE(R,STACK,STACK),.
        CALL EXFREE(NIL,NIL,P),.ALINK(R)=P,.
        ETAT=0,.END,.END,.
        ELSE DO,.
        IF 'A' LE S AND S LE '9' THEN DO,.
        IF S GE'0' THEN CALL LIRNBR,.ELSE CALL LIRATOM,.
        IF ETAT=1 THEN DO,.
        CALL EXFREE(SO,NIL,R),.BLINK(P)=R,.P=R,.END,.
        ELSE DO,.
        ALINK(P)=SO,.ETAT=1,.END,.END,.
        ELSE DO,.
        IF S='*' OR S='-' OR S='+'OR S='/' THEN DO,.
        IF S='*' THEN DO,.
        K=K+1,.CALL LIRSYMB,.
        IF S='*' THEN CALL EXFREE(EXPT,NIL,R),.
        ELSE DO,.K=K-1,.CALL EXFREE(TIMES,NIL,R),.END,.
        END,.
```

```
SPPL1..PROCEDURE OPTIONS(MAIN),.

          ELSE DO,.
          IF S='+' THEN CALL EXFREE(PLUS,NIL,R),.
          ELSE DO,.
          IF S='-' THEN CALL EXFREE(DIFFERENCE,NIL,R),.
          ELSE CALL EXFREE(QUOTIENT,NIL,R),.END,.END,.
          BLINK(P)=R,.P=R,.END,.
          ELSE DO,.
          IF S=')' THEN DO,.
          CALL POP1(P,STACK),.
          IF P=NIL THEN RETURN,.
          IF ETAT=0 THEN DO,.ALINK(P)=NIL,.ETAT=1,.END,.END,.
          END,.END,.END,.
          GOTO B,.
          END LIRFORM,.
     LIREXPR..PROCEDURE,.
          K=80,.STACK=NIL,.INDIC=1,.
     A..  CALL LIRSYMB,.
          IF S='(' THEN DO,.
          CALL EXFREE(NIL,NIL,QO),.P=QO,.
          CALL EXFREE(NIL,STACK,STACK),.
          ETAT=0,.END,.
          ELSE DO,.K=K+1,.GOTO A,.END,.
     B..  K=K+1,.CALL LIRSYMB,.
          IF S='('THEN DO,.
          IF ETAT=0 THEN DO,.
          CALL EXFREE(NIL,NIL,R),.ALINK(P)=R,.
          CALL EXFREE(P,STACK,STACK),.P=R,.END,.
          ELSE DO,.
          CALL EXFREE(NIL,NIL,R),.BLINK(P)=R,.
          CALL EXFREE(R,STACK,STACK),.
          CALL EXFREE(NIL,NIL,P),.ALINK(R)=P,.
          ETAT=0,.END,.END,.
          ELSE DO,.
          IF 'A' LE S AND S LE '9' THEN DO,.
          IF S GE '0' THEN CALL LIRNBR,.ELSE CALL LIRATOM,.
          IF ETAT =1 THEN DO,.
          CALL EXFREE(SO,NIL,R),.BLINK(P)=R,.P=R,.END,.
          ELSE DO,.
          ALINK(P)=SO,.ETAT=1,.END,.END,.
          ELSE DO,.
          IF S='.' THEN DO,.
          IF ETAT=1 THEN DO,.
     BA.. K=K+1,.CALL LIRSYMB,.
          IF 'A' LE S AND S LE '9' THEN DO,.
          IF S GE '0' THEN CALL LIRNBR,.ELSE CALL LIRATOM,.
          BLINK(P)=SO,.END,.
          ELSE DO,.
          IF S='(' THEN DO,.
          CALL EXFREE(NIL,NIL,R),.BLINK(P)=R,.P=R,.ETAT=0,.
          CALL EXFREE(P,STACK,STACK),.END,.
          ELSE GOTO BA,.END,.END,.
          ELSE DO,.
          CALL PRINTCOMMENT('   ERREUR A LA LECTURE        '),.
          QO=NIL,.RETURN,.END,.END,.
          ELSE DO,.
          IF S=')' THEN DO,.
          CALL POP1(P,STACK),.
          IF P=NIL THEN RETURN,.
          IF ETAT=0 THEN DO,.ALINK(P)=NIL,.ETAT=1,. END,.END,.
          END,.END,.END,.
          GOTO B,.
          END LIREXPR,.
```

```
SPPL1..PROCEDURE OPTIONS(MAIN),.

      LIRNBR..PROCEDURE,.
            I=0,.
      NBR1..S=SUBSTR(BUF1,K,1),.
            IF S GE'0' AND S LE'9' THEN DO,.
            I=10*I+S,.K=K+1,.GOTO NBR1,.END,.
            K=K-1,.I=-I-1,.RS=LISTN,.
      NBR2..IF RS NE NIL THEN DO,.
            SO=ALINK(RS),.N=ALINK(SO),.
            IF ALINK(N) =I THEN RETURN,.
            RS=BLINK(RS),.GOTO NBR2,.END,.
            CALL EXFREE(I,NIL,SO),.
            CALL EXFREE(SO,NUMBER,SO),.
            CALL EXFREE(SO,LISTN,LISTN),.
            ALINK(PPO)=LISTN,.
            END LIRNBR,.
      LIRATOM..PROCEDURE,.
            L=K,.
      A..   K=K+1,.S=SUBSTR(BUF1,K,1),.
            IF 'A' LE S AND S LE '9' THEN GOTO A,.
            M=K-L,.K=K-1,.N=(M+1)/2,.
            IF M LE 2 THEN RS=LISTI(1),.
            ELSE DO,.
            IF M GE 7 THEN RS=LISTI(6),.
            ELSE RS=LISTI(M-1),.END,.
            DO I=1 TO N,.
            BUF(I)=SUBSTR(BUF1,L,2),.L=L+2,.END,.
            IF M NE (2*N) THEN SUBSTR(BUF(N),2,1)=' ',.
      C..   SO,R=ALINK(RS),.
            R=ALINK(R),.I=1,.
      CA..  L=BLINK(R),.
            IF INFO(R)=BUF(I) THEN DO,.
            IF I=N THEN DO,.
            IF L=NIL THEN RETURN,.END,.
            ELSE DO,.
            IF L NE NIL THEN DO,.
            I=I+1,.R=L,.GOTO CA,.END,.END,.END,.
            R=BLINK(RS),.
            IF R NE NIL THEN DO,.RS=R,.GOTO C,.END,.
            SO=NIL,.
            DO I=N TO 1 BY -1,.
            CALL EXFREE(BOF(I),SO,SO),.END,.
            CALL EXFREE(SO,NIL,SO),.
            CALL EXFREE(SO,NIL,R),.
            BLINK(RS)=R,.
            END LIRATOM,.
      LIRSYMB..PROCEDURE,.
            IF K GE 80 THEN DO,.CALL READ,.K=1,.END,.
            S=SUBSTR(BUF1,K,1),.
            END LIRSYMB,.
      READ..PROCEDURE,.
            READ FILE(INPUT) INTO(BUF1),.
            SUBSTR(BUF2,2)=BUF1,.CALL EDIT,.
            END READ,.
      EVALQUOTE..PROCEDURE,.
            DCL LABEL(45) LABEL,.
            DCL FLABEL(9) LABEL,.
            DCL SLABEL(6) LABEL,.
            DCL LABX LABEL,.
            ALIST,GOLIST,STACK=NIL,.
            CALL PRINTCOMMENT('   DEBUT DE EVALQUOTE          '),.
            PHI=ALINK(QQ),.
            IF PHI GT 50 AND PHI LT 70 THEN DO,.
```

```
SPPL1..PROCEDURE OPTIONS(MAIN),.

        EPSI=QO,.GOTO EVAL,.END,.
        R=BLINK(PHI),.
        IF ALINK(R)=FEXPR THEN DO,.
        PHI=BLINK(R),.
        CALL EXFREE(NIL,NIL,LIST) ,.
        CALL EXFREE(BLINK(QO),LIST,LIST),.END,.
        ELSE LIST=BLINK(QO),.
APPLY..INDIC=3,.
        IF PHI LE 50 THEN GOTO LABEL(PHI),.
        ACCU=ALINK(PHI),.
        IF ALINK(ACCU) LT 0 THEN GOTO APPLY2,.
        CALL EXFREE(ALIST,STACK,STACK),.
        CALL EXFREE(RETOUR5,STACK,STACK),.
        IF ACCU=LAMBDA THEN GOTO APPLY3,.
        IF ACCU=LABELL THEN GOTO APPLY4,.
        IF ACCU=FUNARG THEN GOTO APPLY5,.
        CALL PRINTCOMMENT('   ERREUR AU DEBUT DE APPLY   '),.
        GOTO ENDEVALQUOTE,.
        /* ATOM */
LABEL(1)..
        ACCU=ALINK(LIST),.ACCU=ALINK(ACCU),.
        IF ALINK(ACCU) LT 0 THEN ACCU=T,. ELSE ACCU=F,.
        GOTO FIN,.
        /* CAR */
LABEL(2)..
        ACCU=ALINK(LIST),.ACCU=ALINK(ACCU),.GOTO FIN,.
        /* CDR */
LABEL(3)..
        ACCU=ALINK(LIST),.ACCU=BLINK(ACCU),.GOTO FIN,.
        /* CONS */
LABEL(4)..
        ACCU=BLINK(LIST),.CALL EXFREE(ALINK(LIST),ALINK(ACCU),ACCU),.
        GOTO FIN,.
        /* EQ */
LABEL(5)..
        ACCU=BLINK(LIST),.
        IF ALINK(LIST)=ALINK(ACCU) THEN ACCU=T,.ELSE ACCU=F,.
        GOTO FIN,.
        /* DEFINE */
LABEL(6)..
        ACCU=ALINK(LIST),.
APPLY16..P=ALINK(ACCU),.R=BLINK(P),.P=ALINK(P),.
        CALL EXFREE(EXPR,ALINK(R),BLINK(P)),.
        ACCU=BLINK(ACCU),.
        IF ACCU NE NIL THEN GOTO APPLY16,.
        GOTO FIN,.
        /* CSET */
LABEL(7)..
        ACCU=ALINK(LIST),.P=BLINK(LIST),.
        CALL EXFREE(APVAL,ALINK(P),BLINK(ACCU)),.
        GOTO FIN,.
        /* APPEND */
LABEL(8)..
        R=ALINK(LIST),.ACCU=BLINK(LIST),.ACCU=ALINK(ACCU),.
        IF R NE NIL THEN DO,.
        CALL EXFREE(ALINK(R),ACCU,P),.PHI=P,.
APPEND1..R=BLINK(R),.
        IF R NE NIL THEN DO,.
        CALL EXFREE(ALINK(R),ACCU,M),.BLINK(P)=M,.P=M,.
        GOTO APPEND1,.END,.
        ELSE ACCU=PHI,.END,.
        GOTO FIN,.
```

```
$PPL1..PROCEDURE OPTIONS(MAIN),.

        /* LENGTH */
    LABEL(9)..
        I=-1,.R=ALINK(LIST),.
    LENGTH1..IF R NE NIL THEN DO,.
        I=I-1,.R=BLINK(R),.GOTO LENGTH1,.END,.
        CALL EXFREE(I,NIL,PHI),.
        CALL EXFREE(PHI,NUMBER,ACCU),.
        GOTO FIN,.
        /* NOT */
    LABEL(10)..
        IF ALINK(LIST)=F THEN ACCU=T,.ELSE ACCU=F,.
        GOTO FIN,.
        /* CAAR */
    LABEL(11)..
        ACCU=ALINK(LIST),.ACCU=ALINK(ACCU),.
        ACCU=ALINK(ACCU),.GOTO FIN,.
        /* CADR */
    LABEL(12)..
        ACCU=ALINK(LIST),.ACCU=BLINK(ACCU),.
        ACCU=ALINK(ACCU),.GOTO FIN,.
        /* CDAR */
    LABEL(13)..
        ACCU=ALINK(LIST),.ACCU=ALINK(ACCU),.
        ACCU=BLINK(ACCU),.GOTO FIN,.
        /* COPY */
    LABEL(14)..
        P=ALINK(LIST),.M=ALINK(P),.
        IF ALINK(M) GT 0 THEN DO,.
        CALL EXFREE(P,STACK,STACK),.
        CALL EXFREE(NIL,NIL,PHI),.
        CALL EXFREE(PHI,STACK,STACK),.
        ACCU,N=PHI,.
        LIST=NIL,.
    COPY1..M=ALINK(P),.R=ALINK(M),.
        IF ALINK(R) GE 0 THEN DO,.
        CALL EXFREE(P,LIST,LIST),.
        CALL EXFREE(N,PHI,PHI),.
        CALL EXFREE(NIL,NIL,K),.
        ALINK(N)=K,.N=K,.P=M,.
        GOTO COPY1,.END,.
        ALINK(N)=M,.
    COPY2..M=BLINK(P),.R=ALINK(M),.
        IF ALINK(R) GE 0 THEN DO,.
        CALL EXFREE(NIL,NIL,K),.
        BLINK(N)=K,.N=K,.P=M,.
        GOTO COPY1,.END,.
        BLINK(N)=M,.
        IF LIST NE NIL THEN DO,.
        CALL POP1(P,LIST),.
        CALL POP1(N,PHI),.
        GOTO COPY2,.END,.
        ELSE DO,.
        STACK=BLINK(STACK),.STACK=BLINK(STACK),.END,.END,.
        ELSE ACCU=P,.
        GOTO FIN,.
        /* CDDR */
    LABEL(15)..
        ACCU=ALINK(LIST),.ACCU=BLINK(ACCU),.
        ACCU=BLINK(ACCU),.GOTO FIN,.
        /* LIST */
    LABEL(16)..
        ACCU=LIST,.GOTO FIN,.
```

```
$PPL1..PROCEDURE OPTIONS(MAIN),.

        /* NULL */
    LABEL(17)..
        IF ALINK(LIST)=NIL THEN ACCU=T,.ELSE ACCU=F,.
        GOTO FIN,.
        /* CADDR */
    LABEL(18)..
        ACCU=ALINK(LIST),.ACCU=BLINK(ACCU),.
        ACCU=BLINK(ACCU),.ACCU=ALINK(ACCU),.
        GOTO FIN,.
        /* EQUAL */
    LABEL(19)..
        P=ALINK(LIST),.N=BLINK(LIST),.N=ALINK(N),.
        LABX=FIN,.
    EQUAL0..M=ALINK(P),.Q=ALINK(N),.
        PR=ALINK(M),.PQ=ALINK(Q),.
        IF PR LT 0 OR PQ LT 0 THEN DO,.
        IF P NE N AND ( BLINK(P) NE NUMBER OR PR NE PQ OR BLINK(N) NE
        NUMBER) THEN ACCU = F,.ELSE ACCU=T,.END,.
        ELSE DO,.
        CALL EXFREE(P,STACK,STACK),.
        CALL EXFREE(N,STACK,STACK),.
        PHI,LIST=NIL,.
    EQUAL1..M=ALINK(P),.J=ALINK(N),.
        R=ALINK(M),.Q=ALINK(J),.
        PR=ALINK(R),.PQ=ALINK(Q),.
        IF PR LT 0 OR PQ LT 0 THEN DO,.
        IF M NE J AND (BLINK(M) NE NUMBER OR PR NE PQ OR BLINK(J) NE
        NUMBER) THEN ACCU=F,.
        ELSE DO,.
    EQUAL2..M=BLINK(P),.J=BLINK(N),.
        R=ALINK(M),.Q=ALINK(J),.
        PR=ALINK(R),.PQ=ALINK(Q),.
        IF PR LT 0 OR PQ LT 0 THEN DO,.
        IF M NE J AND (BLINK(M) NE NUMBER OR PR NE PQ OR BLINK(J) NE
        NUMBER) THEN ACCU=F,.
        ELSE DO,.
        IF LIST=NIL THEN ACCU=T,.
        ELSE DO,.
        CALL POP1(P,LIST),.
        CALL POP1(N,PHI),.
        GOTO EQUAL2,.END,.END,.END,.
        ELSE DO,.
        P=M,.N=J,.GOTO EQUAL1,.END,.END,.END,.
        ELSE DO,.
        CALL EXFREE(P,LIST,LIST),.CALL EXFREE(N,PHI,PHI),.
        P=M,.N=J,.GOTO EQUAL1,.END,.
        STACK=BLINK(STACK),.STACK=BLINK(STACK),.
        END,.GOTO LABX,.
        /* RPLACA */
    LABEL(20)..
        ACCU=ALINK(LIST),.Q=BLINK(LIST),.
        ALINK(ACCU)=ALINK(Q),.GOTO FIN,.
        /* RPLACD */
    LABEL(21)..
        ACCU=ALINK(LIST),.Q=BLINK(LIST),.
        BLINK(ACCU)=ALINK(Q),.GOTO FIN,.
        /* MEMBER */
    LABEL(22)..
        I=ALINK(LIST),.K=BLINK(LIST),.K=ALINK(K),.
        CALL EXFREE(K,STACK,STACK),.
        ACCU=F,.LABX=MEMBER2,.
    MEMBER1..IF   K NE NIL THEN DO,.
```

```
PPL1..PROCEDURE OPTIONS(MAIN),.

        P=I,.N=ALINK(K),.K=BLINK(K),.
        GOTO EQUAL0,.
MEMBER2..IF ACCU=F THEN GOTO MEMBER1,.END,.
        STACK=BLINK(STACK),.GOTO FIN,.
        /* RECLAIM */
LABEL(23)..
        ALINK(97)=STACK,.ALINK(98)=ALIST,.ALINK(99),ALINK(100)=NIL,.
        CALL GARBCOL,.GOTO FIN,.
        /* REVERSE */
LABEL(24)..
        PHI=NIL,.LIST=ALINK(LIST),.
REVERSE1..IF LIST NE NIL THEN DO,.
        CALL EXFREE(ALINK(LIST),PHI,PHI),.LIST=BLINK(LIST),.
        GOTO REVERSE1,.END,.
        ACCU=PHI,.GOTO FIN,.
        /* GO */
LABEL(25)..
        P=GOLIST,.Q=ALINK(LIST),.ACCU=NIL,.
GO1.. IF P=NIL THEN
        CALL PRINTCOMMENT('   ETIQUETTE INEXISTANTE      '),.
        ELSE DO,.
        R=ALINK(P),.
        IF ALINK(R)=Q THEN DO,.
        J=BLINK(R),.
        IF ALINK(STACK)=RETOUR6 THEN GOTO PROG5,.
        CALL PRINTCOMMENT('   EMPLOI ILLICITE DE GO      '),.
        END,.
        ELSE DO,.
        P=BLINK(P),.GOTO GO1,.END,.END,.
        GOTO ENDEVALQUOTE,.
        /* DUP */
LABEL(26)..
        P=ALINK(LIST),.P=ALINK(P),.
        CALL EXFREE(ALINK(P),NIL,PHI),.
        CALL EXFREE(PHI,NUMBER,ACCU),.
        GOTO FIN,.
        /* SET */
LABEL(27)..
        P=ALIST,.ACCU=ALINK(LIST),.
        J=BLINK(LIST),.
SET1..IF P = NIL THEN DO,.
        CALL PRINTCOMMENT('   VARIABLE INEXISTANTE       '),.
        GOTO ENDEVALQUOTE,.END,.
        R=ALINK(P),.
        IF ALINK(R)=ACCU THEN DO,.
        ACCU,BLINK(R)=ALINK(J),.GOTO FIN,.END,.
        P=BLINK(P),.GOTO SET1,.
        /* ADD1 */
LABEL(28)..
        ACCU=ALINK(LIST),.P=ALINK(ACCU),.
        ALINK(P)=ALINK(P)-1,.GOTO FIN,.
        /* SUB1 */
LABEL(29)..
        ACCU=ALINK(LIST),.P=ALINK(ACCU),.
        ALINK(P)=ALINK(P)+1,.GOTO FIN,.
        /* PLUS */
LABEL(30)..
        I=0,.
PLUS1..IF LIST=NIL THEN GOTO FINARITH,.
        P=ALINK(LIST),.P=ALINK(P),.I=I-ALINK(P)-1,.
        LIST=BLINK(LIST),.GOTO PLUS1,.
        /* TIMES */
```

```
PPL1..PROCEDURE OPTIONS(MAIN),.

      LABEL(32)..
            I=1,.
      TIMES1..IF LIST=NIL THEN GOTO FINARITH,.
            P=ALINK(LIST),.P=ALINK(P),.I=(-ALINK(P)-1)*I,.
            LIST=BLINK(LIST),.GOTO TIMES1,.
      FINARITH..I=-I-1,.
            CALL EXFREE(I,NIL,PHI),.
            CALL EXFREE(PHI,NUMBER,ACCU),.
            GOTO FIN,.
            /* LESSP */
      LABEL(33)..
            P=ALINK(LIST),.LIST=BLINK(LIST),.R=ALINK(LIST),.
      LESSP1..P=ALINK(P),.R=ALINK(R),.
            IF ALINK(P) GT ALINK(R) THEN ACCU=T,.ELSE ACCU=F,.
            GOTO FIN,.
            /* ZEROP */
      LABEL(34)..
            R=ALINK(LIST),.R=ALINK(R),.
            IF ALINK(R)=-1 THEN ACCU=T,.ELSE ACCU=F,.
            GOTO FIN,.
            /* PRINT */
      LABEL(35)..
            CALL EXFREE(STACK,ALIST,ACCU),.
            PO=ALINK(LIST),.
            CALL PRINTEXPR,.
            STACK=ALINK(ACCU),.ALIST=BLINK(ACCU),.
            ACCU=PO,.GOTO FIN,.
            /* EVAL */
      LABEL(31)..
            EPSI=ALINK(LIST),.P=BLINK(LIST),.
            IF P NE NIL THEN DO,.
            CALL EXFREE(ALIST,STACK,STACK),.
            CALL EXFREE(RETOUR5,STACK,STACK),.
            ALIST=ALINK(P),.END,.
            GOTO EVAL,.
            /* APPLY */
      LABEL(36)..
            PHI=ALINK(LIST),.LIST=BLINK(LIST),.
            P=BLINK(LIST),.
            IF P NE NIL THEN DO,.
            CALL EXFREE(ALIST,STACK,STACK),.
            CALL EXFREE(RETOUR5,STACK,STACK),.
            ALIST=ALINK(P),.END,.
            LIST=ALINK(LIST),.GOTO APPLY,.
            /* RETURN */
      LABEL(37)..
            ACCU=ALINK(LIST),.
            IF ALINK(STACK)=RETOUR5 THEN DO,.
            P=BLINK(STACK),.GOTO PROG7,.END,.
            CALL PRINTCOMMENT('   EMPLOI ILLICITE DE RETURN  '),.
            GOTO ENDEVALQUOTE,.
            /* QUOTIENT */
      LABEL(38)..
            I=ALINK(LIST),.I=ALINK(I),.
            I=-ALINK(I)-1,.LIST=BLINK(LIST),.
            P=ALINK(LIST),.P=ALINK(P),.
            I=I/(-ALINK(P)-1),.GOTO FINARITH,.
            /* NUMBERP  */
      LABEL(39)..
            ACCU=ALINK(LIST),.
            IF BLINK(ACCU)=NUMBER THEN ACCU=T,.ELSE ACCU=F,.
            GOTO FIN,.
```

```
PPL1..PROCEDURE OPTIONS(MAIN),.

        /* DIFFERENCE */
    LABEL(40)..
        I=ALINK(LIST),.I=ALINK(I),.
        I=-ALINK(I),.LIST=BLINK(LIST),.
        P=ALINK(LIST),.P=ALINK(P),.
        I=I+ALINK(P),.GOTO FINARITH,.
        /* GREATERP */
    LABEL(41)..
        R=ALINK(LIST),.LIST=BLINK(LIST),.
        P=ALINK(LIST),.GOTO LESSP1,.
        /* PRINTFORM */
    LABEL(42)..
        CALL EXFREE(STACK,ALIST,ACCU),.
        PO=ALINK(LIST),.
        CALL PRINTFORM,.
        STACK=ALINK(ACCU),.ALIST=BLINK(ACCU),.
        ACCU=PO,.
        GOTO FIN,.
        /* LIRFORM */
    LABEL(43)..
        CALL EXFREE(STACK,ALIST,ACCU),.
        CALL LIRFORM,.
        STACK=ALINK(ACCU),.ALIST=BLINK(ACCU),.
        ACCU=QO,.
        GOTO FIN,.
        /* EXPT */
    LABEL(44)..
        I=1,.
        L=ALINK(LIST),.L=ALINK(L),.L=-ALINK(L)-1,.
        LIST=BLINK(LIST),.
        J=ALINK(LIST),.J=ALINK(J),.J=-ALINK(J)-1,.
        DO K=1 TO J,.I=I*L,.END,.
        GOTO FINARITH,.
        /* MINUS  */
    LABEL(45)..
        LIST=ALINK(LIST),.
        IF ALINK(LIST)=MINUS THEN DO,.
        ACCU=BLINK(LIST),.ACCU=ALINK(ACCU),.END,.
        ELSE DO,.
        CALL EXFREE(LIST,NIL,LIST),.
        CALL EXFREE(MINUS,LIST,ACCU),.END,.
        GOTO FIN,.
        /* AUTRE ATOME */
    APPLY2..
        ACCU=BLINK(PHI),.
        IF ALINK(ACCU)=EXPR THEN PHI=BLINK(ACCU),.
        ELSE DO,.
        ACCU=ALIST,.
    APPLY21..Q=ALINK(ACCU),.
        IF ALINK(Q)=PHI THEN PHI=BLINK(Q),.
        ELSE DO,.
        ACCU=BLINK(ACCU),.
        IF ACCU NE NIL THEN GOTO APPLY21,.
        CALL PRINTCOMMENT('   FONCTION NON DEFINIE        '),.
        ACCU=PHI,.
        GOTO ENDEVALQUOTE,.END,.END,.
        GOTO APPLY,.
        /* LAMBDA */
    APPLY3..
        EPSI=BLINK(PHI),.R=ALINK(EPSI),.
    APPLY31..IF R=NIL OR LIST=NIL THEN DO,.
        EPSI=BLINK(EPSI),.EPSI=ALINK(EPSI),.GOTO EVAL,.END,.
```

```
PPLI..PROCEDURE OPTIONS(MAIN),.

        CALL EXFREE(NIL,ALIST,ALIST),.
        CALL EXFREE(ALINK(R),ALINK(LIST),ALINK(ALIST)),.
        LIST=BLINK(LIST),.R=BLINK(R),.
        GOTO APPLY31,.
        /* LABEL */
    APPLY4..
        P=BLINK(PHI),.R=BLINK(P),.R=ALINK(R),.
        CALL EXFREE(NIL,ALIST,ALIST),.
        CALL EXFREE(ALINK(P),R,ALINK(ALIST)),.PHI=R,.
        GOTO APPLY,.
        /* FUNARG */
        APPLY5..
        P=BLINK(PHI),.PHI=ALINK(P),.P=BLINK(P),.
        ALIST=ALINK(P),.GOTO APPLY,.
    EVAL..INDIC=4,.ACCU=ALINK(EPSI),.
        IF ACCU GT 50 THEN DO,.
        IF ACCU LT 70 THEN GOTO FLABEL(ACCU-50),.
        IF ALINK(ACCU) LT 0 THEN DO,.
        ACCU=BLINK(EPSI),.
    EVAL21..IF ACCU=NIL THEN DO,.
        ACCU=ALIST,.
    EVAL22..Q=ALINK(ACCU),.
        IF ALINK(Q)=EPSI THEN ACCU=BLINK(Q),.
        ELSE DO,.
        ACCU=BLINK(ACCU),.
        IF ACCU NE NIL THEN GOTO EVAL22,.
        CALL PRINTCOMMENT('   VARIABLE SANS VALEUR       '),.
        ACCU=EPSI,.
        GOTO ENDEVALQUOTE,.END,.END,.
        ELSE DO,.
        IF ACCU=NUMBER THEN ACCU=EPSI,.
        ELSE DO,.
        IF ALINK(ACCU)=APVAL THEN ACCU=BLINK(ACCU),.
        ELSE DO,.
        ACCU=NIL,.GOTO EVAL21,.END,.END,. END,.
        GOTO FIN,.END,.
        R=BLINK(ACCU),.
        IF ALINK(R)=FEXPR THEN DO,.
        PHI=BLINK(R),.
        CALL EXFREE(ALIST,NIL,ACCU),.
        CALL EXFREE(BLINK(EPSI),ACCU,LIST),.
        GOTO APPLY,.END,.END,.
        /* APPLICATION DE FONCTION */
     EVAL4..
        PHI=ALINK(EPSI),.LIST=BLINK(EPSI),.
        IF LIST=NIL THEN GOTO APPLY,.
        CALL EXFREE(NIL,NIL,ACCU),.
        CALL EXFREE(ACCU,STACK,STACK),.
        CALL EXFREE(PHI,STACK,STACK),.
        CALL EXFREE(BLINK(LIST),STACK,STACK),.
        CALL EXFREE(ACCU,STACK,STACK),.
        CALL EXFREE(RETOUR4,STACK,STACK),.
        EPSI=ALINK(LIST),.GOTO EVAL,.
     SLABEL(4)..
        P=BLINK(STACK),.M=ALINK(P),.ALINK(M)=ACCU,.
        ACCU=BLINK(P),.R=ALINK(ACCU),.
        IF R=NIL THEN DO,.
        PHI=BLINK(ACCU),.LIST=BLINK(PHI),.STACK=BLINK(LIST),.
        PHI=ALINK(PHI),.LIST=ALINK(LIST),.GOTO APPLY,.END,.
        ALINK(ACCU)=BLINK(R),.EPSI=ALINK(R),.
        CALL EXFREE(NIL,NIL,N),.BLINK(M)=N,.ALINK(P)=N,.
        GOTO EVAL,.
```

```
PPL1..PROCEDURE OPTIONS(MAIN),.

        /* OR */ /* AND */
    FLABEL(1)..
    FLABEL(2)..
        ACCU=ACCU-50,.
        R=BLINK(EPSI),.
        CALL EXFREE(BLINK(R),STACK,STACK),.
        CALL EXFREE(ACCU,STACK,STACK),.
        EPSI=ALINK(R),.GOTO EVAL,.
    SLABEL(1)..
        EPSI=BLINK(STACK),.
        IF ACCU=F THEN DO,.
        R=ALINK(EPSI),.
        IF R NE NIL THEN DO,.,.
        ALINK(EPSI)=BLINK(R),.EPSI=ALINK(R),.
        GOTO EVAL,.END,.END,.
        ELSE ACCU=T,.
        STACK=BLINK(EPSI),.GOTO FIN,.
    SLABEL(2)..
        EPSI=BLINK(STACK),.
        IF ACCU NE F THEN DO,.
        R=ALINK(EPSI),.
        IF R NE NIL THEN DO,.
        ALINK(EPSI)=BLINK(R),.EPSI=ALINK(R),.
        GOTO EVAL,.END,.
        ELSE ACCU=T,.END,.
        STACK=BLINK(EPSI),.GOTO FIN,.
        /* CSETQ */
    FLABEL(7)..
        ALINK(EPSI)=CSET,.GOTO QUOTE1,.
        /* GOQ */
    FLABEL(3)..
        ALINK(EPSI)=GO,.GOTO QUOTE1,.
        /* SETQ */
    FLABEL(4)..
        ALINK(EPSI)=SET,.
    QUOTE1..R=BLINK(EPSI),.
        CALL EXFREE(ALINK(R),NIL,ACCU),.
        CALL EXFREE(QUOTE,ACCU,ALINK(R)),.
        GOTO EVAL4,.
        /* COND */
    FLABEL(5)..
        EPSI=BLINK(EPSI),.
        CALL EXFREE(BLINK(EPSI),STACK,STACK),.
        ACCU=ALINK(EPSI),.EPSI=ALINK(ACCU),.ACCU=BLINK(ACCU),.
        CALL EXFREE(ALINK(ACCU),STACK,STACK),.
        CALL EXFREE(RETOUR3,STACK,STACK),.
        GOTO EVAL,.
    SLABEL(3)..
        IF ACCU NE F THEN DO,.
        STACK=BLINK(STACK),.
        CALL POP1(EPSI,STACK),.
        STACK=BLINK(STACK),.END,.
        ELSE DO,.
        ACCU=BLINK(STACK),.EPSI=BLINK(ACCU),.P=ALINK(EPSI),.
        IF P=NIL THEN DO,.
        STACK=BLINK(EPSI),.ACCU=NIL,.GOTO FIN,.END,.
        ALINK(EPSI)=BLINK(P),.P=ALINK(P),.Q=BLINK(P),.
        ALINK(ACCU)=ALINK(Q),.EPSI=ALINK(P),.END,.
        GOTO EVAL,.
        /* QUOTE */
    FLABEL(6)..
        ACCU=BLINK(EPSI),.ACCU=ALINK(ACCU),.GOTO FIN,.
```

```
PL1..PROCEDURE OPTIONS(MAIN),.

      /* FONCTION */
FLABEL(8)..
      EPSI=BLINK(EPSI),.
      CALL EXFREE(ALIST,NIL,ACCU),.
      CALL EXFREE(ALINK(EPSI),ACCU,ACCU),.
      CALL EXFREE(FUNARG,ACCU,ACCU),.
      GOTO FIN,.
      /* PROG */
FLABEL(9)..
      CALL EXFREE(GOLIST,STACK,STACK),.
      CALL EXFREE(ALIST,STACK,STACK),.
      R=BLINK(EPSI),.P=ALINK(R),.J=BLINK(R),.ACCU=GOLIST,.
PROG1..IF P NE NIL THEN DO,.
      CALL EXFREE(NIL,ALIST,ALIST),.
      CALL EXFREE(ALINK(P),NIL,ALINK(ALIST)),.
      P=BLINK(P),.GOTO PROG1,.END,.
      R=J,.
PROG3..IF R NE NIL THEN DO,.
      P=ALINK(R),.I=ALINK(P),.R=BLINK(R),.
      IF ALINK(I) LT 0 THEN DO,.
      CALL EXFREE(NIL,ACCU,ACCU),.
      CALL EXFREE(P,R,ALINK(ACCU)),.END,.
      GOTO PROG3,.END,.
      GOLIST=ACCU,.
      CALL EXFREE(ACCU,STACK,STACK),.
      CALL EXFREE(BLINK(J),STACK,STACK),.
      CALL EXFREE(RETOUR6,STACK,STACK),.
PROG5..EPSI=ALINK(J),.R=ALINK(EPSI),.
      IF ALINK(R) LT 0 THEN DO,.
      J=BLINK(J),.GOTO PROG5,.END,.
      R=BLINK(STACK),.ALINK(R)=BLINK(J),.GOTO EVAL,.
SLABEL(6)..
      P=BLINK(STACK),.J=ALINK(P),.
      IF J NE NIL THEN GOTO PROG5,.
PROG7..STACK=BLINK(P),.STACK=BLINK(STACK),.
      CALL POP1(ALIST,STACK),.
      CALL POP1(GOLIST,STACK),.
      GOTO FIN,.
      /* GESTION DU STACK */
SLABEL(5)..
      STACK=BLINK(STACK),.
      CALL POP1(ALIST,STACK),.
FIN..
      IF STACK=NIL THEN,.
      ELSE DO,.
      Q=ALINK(STACK),.GOTO SLABEL(Q),.END,.
ENDEVALQUOTE..
      CALL PRINTCOMMENT('   FIN DE EVALQUOTE          '),.
      END EVALQUOTE,.
FINLISP..CLOSE FILE(INPUT),.CLOSE FILE(PRINT),.
      END LISPPL1,.
```

BIBLIOGRAPHIE.

[1] D. Ribbens, Programmation non numérique, LISP 1.5, Dunod, Paris(1969).

[2] J. Mc Carthy et autres, LISP 1.5 Programmer's Manual, The M.I.T. Press, Cambridge, Mass.(1962).

[3] H. Leroy, Programmation fonctionnelle, LISP, notes de cours, F.N.D.P., Namur(1972).

[4] E. Berkeley, D. Bobrow (Eds), The programming language LISP : Its Operations and Applications, Information International, Inc., Cambridge, Mass.(1964).

[5] C. Weissman, LISP 1.5 Primer, Dickenson Publishing Company, Inc., Belmont, California(1967).

[6] T.P. Hart and T.G. Evans, Notes on Implementing LISP for the M-460 Computer, dans 4 .

[7] P.L. Wodon, A LISP 1.5 Interpreteur for the P3 Computer, M.B.L.E. Laboratoires de recherches, Report R52, (1966).

[8] D.E. Knuth, The Art of Computer Programming, Vol. 1 : Fundamental Algorithms, Addison-Wesley.

[9] B. Wegbreit, A generalised compactifying garbage collector, The Computer Journal, Vol.15, Numb.3.